AFRL-RI-RS-TR-2015-157

ADVANCED DEVELOPMENT OF CERTIFIED OS KERNELS

YALE UNIVERSITY

JUNE 2015

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-157 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUNE 2015	FINAL TECHNICAL REPORT	SEP 2010 – DEC 2014

4. TITLE AND SUBTITLE

ADVANCED DEVELOPMENT OF CERTIFIED OS KERNELS

5a. CONTRACT NUMBER
FA8750-10-2-0254

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Zhong Shao

5d. PROJECT NUMBER
CRSH

5e. TASK NUMBER
YA

5f. WORK UNIT NUMBER
LE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Yale University
105 Wall Street
New Haven, CT 06511-6614

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2015-157

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2015-3064
Date Cleared: 16 Jun 15

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The PI and his team at Yale have successfully developed (1) a clean-slate CertiKOS hypervisor kernel that runs on multicore platforms and supports Linux and ROS applications; (2) a new certified programming methodologies and tools that can verify contextual correctness, liveness, and security properties in a unified setting; (3) a fully verified single-core CertiKOS in Coq; (4) new semantics and logics for reasoning about information flow control with declassification, resource analysis, and fine-grained concurrent programs; and (5) new proof assistant language VeriML and Coq Ltac libraries.

15. SUBJECT TERMS

Certified Software; Certified OS Kernels; Certified Compilers; Abstraction Layers; Modularity; Deep Specifications; Program Verification; Certified Resource Bound Analysis; Concurrency; Relay-Guarantee Reasoning; Simulation and Refinement; Termination Preservation; Information Flow Control; Liveness Properties; Quantitative Verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILMAR W. SIFRE**
U	U	U	UU	503	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

List of Figures

1 SUMMARY

OS kernels form the backbone of all system software. They can have the greatest impact on the resilience, extensibility, and security of today's computing hosts. Recent effort on seL4 has demonstrated the feasibility of building large scale formal proofs of functional correctness for a general-purpose microkernel, but the cost of such verification is still prohibitive, and it is unclear how to use such a verified kernel to reason about user-level programs and other kernel extensions.

Under this DARPA CRASH effort (FA8750-10-2-0254), the PI (Principal Investigator) and his team has developed a clean-slate CertiKOS hypervisor kernel that runs on Intel and AMD multicore platforms with hardware virtualization and can boot Linux and ROS applications in its multiple virtual machines. A version of CertiKOS is now deployed on all the ground vehicle platforms (LandShark UGV and American Built Car) in the DARPA HACMS program.

The PI and his team have also developed a new set of certified programming methodologies and tools that support programming and composing certified abstraction layers (in C or assembly) and can verify contextual safety, correctness, liveness, and security properties in one unified setting.

Using these new languages and tools, they developed a new compositional architecture for building certified OS kernels. Because the very purpose of an OS kernel is to build layers of abstraction over hardware resources, they insisted on uncovering and specifying these layers formally, and then verifying each kernel module at its proper abstraction level. To support reasoning about user-level programs and linking with other certified kernel extensions, they proved a strong contextual refinement property for every kernel function, which states that the implementation of each such function will behave like its specification under any kernel/user (or host/guest) context. To demonstrate the effectiveness of this new approach, they have successfully specified a uniprocessor variant of their full CertiKOS kernel and verified its (contextual) functional correctness property in the Coq proof assistant. They showed how to extend their base kernel with new features such as virtualization and ring-0 processes and how to quickly adapt existing verified layers to build new certified kernels for different domains. Their certified hypervisor OS kernel is written in 5500 lines of C and x86 assembly, and can successfully boot a version of Linux as a guest. The entire specification and proof effort took less than 1.5 person years.

They have also developed new semantics and logics for supporting Declarative Decentralized Information Flow Control (DIFC) with declassification. They proposed a new framework which advocate the use of an instrumented semantics for reasoning and the erasure semantics for execution. Their new program logic can be used tor verify security properties for low-level C or assembly programs. They showed that they can prove a new form of non-interference properties even in the presence of declassification. This technology is now being ported into their CertiKOS kernels.

They have also developed new ground-breaking certified resource analysis tools and new logics for verifying safety and liveness of fine-grained shared memory concurrent programs.

On the formal methods side, they have also made the first comprehensive study that aims to address the architecture deficiencies in all of today's proof assistants. They proposed a new proof-assistant architecture that uses extensible conversion rules and static proof expressions to support effective and principled proof development. They developed the design, the complete meta theory, and a full compiler of a novel programming language called VeriML which realizes the new architecture and also offers a unified platform for coding all kinds of computation on logical terms.

2 INTRODUCTION

Operating System (OS) kernels and hypervisors form the backbone of every safety-critical software system in the world. Hence it is highly desirable to formally verify the correctness of these programs [35]. Recent work on seL4 [19, 20] has shown that it is feasible to formally prove the functional correctness property of a general-purpose microkernel, but the cost of such verification is still quite prohibitive. It took the seL4 team more than 11 person years (effort for tool development excluded) to verify 7500 lines of sequential C code, yet the resulting kernel still contains 1200 lines of additional C code and 600 lines of assembly code that are not verified. Worse still, even after all these efforts, the current verified seL4 kernel cannot be used to reason about user-level programs as it does not verify important features such as virtual-memory page faults and address translation.

What makes the verification of OS kernels so challenging?

First, OS kernels are complex artifacts; they contain many interdependent components that are difficult to untangle. Their invariants can involve machine level details (e.g., how the virtual memory hardware works) but can also cut across multiple abstraction boundaries (e.g., different views of an address space under kernel/user or host/guest modes). Several researchers [1, 41] observed that even writing down a good and easy-to-maintain formal specification alone is already a major roadblock for any such verification effort.

Second, OS kernels are often written in C, which only supports limited forms of abstraction. Verification of C programs is especially hard if they manipulate low-level data structures (e.g., thread queues, allocation tables). The seL4 effort used an intermediate executable specification (derived from a Haskell prototype) to hide some messy C specifics, but this alone is not enough for enforcing abstraction among different kernel components; seL4 had to introduce capabilities which add significant implementation complexities to the kernel.

Third, OS kernels are developed for managing and multiplexing hardware, so it is important to have a machine model that can describe hardware details. The C language is too high level for this purpose. For example, while most kernel code can be written in C, many key kernel concepts (e.g., context switches, address translation, page fault handling) can only be given accurate semantics at the assembly level. Consequently, we need a formal assembly model to define many kernel behaviors, but we also want to verify most kernel code at a much higher abstraction level.

Fourth, OS kernel verification would not scale if it does not support extensibility. One advantage of a verified kernel is the existence of formal specifications for all of its components. In theory, this would allow us to add certified kernel plug-ins [36] as long as they do not violate any existing kernel invariants. In practice, however, if we are unable to decompose kernel invariants into small independent pieces, even modifying an existing (or adding a new) verified component may force us to rewrite the proofs for the entire kernel.

Under this DARPA CRASH (Clean-Slate Design of Resilient, Adaptive, Secure Hosts) effort

(FA8750-10-2-0254), the PI and his team at Yale University have developed a novel compositional approach that successfully tackles all of the above challenges in building certified OS kernels. They believe that, to make verification scale and to provide strong support to extensibility, they must first have a *compositional* specification that can untangle *all* the kernel interdependencies. Because the very purpose of an OS kernel is to build layers of abstraction over bare machines, they insist on meticulously uncovering and specifying these layers (done in the Coq proof assistant [40]), and then verifying each kernel module at its *proper* abstraction level.

The functional correctness of an OS kernel (as done in seL4) is usually stated as a *refinement* property. Roughly speaking, if M_C stands for the C/assembly implementation of a kernel, M_A for its abstract functional specification, and $[\![\cdot]\!]$ for each's corresponding state machine, then M_C refines M_A if there exists a *forward simulation* [28] from $[\![M_C]\!]$ to $[\![M_A]\!]$ (denoted as $[\![M_C]\!] \sqsubseteq [\![M_A]\!]$). Through such refinement, Gerwin *et al* [19, 30, 33, 34] claimed that many properties established for M_A (e.g., confidentiality [30] when M_A is deterministic) can be transferred to M_C.

This claim, unfortunately, fails to hold in the context of any interesting user-level programs. If P stands for a collection of user-level processes and $\bowtie$ for a linking operator, then from $[\![M_C]\!] \sqsubseteq [\![M_A]\!]$ alone, we cannot derive $[\![M_C \bowtie P]\!] \sqsubseteq [\![M_A \bowtie P]\!]$. This is because the semantics of running P on top of M_A (where virtual memory hardware is hidden) is different from that of running P on top of M_C (where page faults and address translation do come into play). Daum *et al* [7] partially closed the gap by extending the original refinement proof to also track memory permissions, but they still did not deal with page faults in their model of user transitions.

Under this new DARPA CRASH effort, the PI and his team instead prove the strong *contextual refinement* property for all kernel modules *directly*: they show that for any kernel/user or host/guest context code P, $[\![M_C \bowtie P]\!] \sqsubseteq [\![M_A \bowtie P]\!]$ always holds. This guarantees that they cannot overlook any subtle difference between machines at different abstraction levels.

More specifically, they developed a new extensible architecture (called CertiKOS) for building certified OS kernels. CertiKOS uses *contextual refinement* as the unifying formalism for composing kernel and user components at different abstraction levels. Each abstraction layer is defined as an assembly-level machine extended with a particular set of abstract states and primitives. However, most of their kernel programs are written in a variant of C (called ClightX) [14], verified at the source level, and compiled and linked together using a modified version [14] of the CompCert verified compiler [21, 22]. CertiKOS is the first architecture that can truly transfer global properties proved for user-level programs (at the kernel specification level) down to the concrete assembly machine level.

Using CertiKOS, they have developed a fully certified **mCertiKOS** kernel in Coq. Unlike seL4, they decompose the specification of mCertiKOS into 33 *logical* abstraction layers, and turn an otherwise prohibitive verification task into many simple and easily automatable sub-tasks. The resulting kernel is a certified assembly implementation that still enjoys a high degree of compositionality. Their layered specification shows that *interdependent* low-level kernel modules

can indeed be untangled and given clear formal semantics.

Using mCertiKOS as the base, we have also built three additional certified kernels: **mCertiKOS-hyp** extends mCertiKOS with virtualization support to form a hypervisor kernel; **mCertiKOS-rz** extends mCertiKOS-hyp with "ring 0" processes (they are "certifiably safe" application programs that can run safely inside the kernel address space, similar to SIPs in Singularity [17]); **mCertiKOS-emb** removes virtual memory and virtualization support from mCertiKOS-rz so that it only supports "ring 0" processes.

They have done a detailed evaluation of their certified development effort, including kernel performance, the cost of layer design and proof development, and the cost of building new extended (or adapted) kernels. All of their certified kernels are practical and can run on stock x86 hardware. Their certified hypervisor kernel (mCertiKOS-hyp) consists of 5500 lines of C and x86 assembly, and can successfully boot a version of Linux as a guest. The entire specification and proof effort took less than 1.5 person years.

Finally, in addition to developing new cutting-edge technologies for building certified OS kernels, the PI and his team have also made significant breakthroughs on the following problems:

- They have developed a clean-slate hypervisor kernel that runs on Intel and AMD multicore platforms with hardware virtualization and can boot Linux and ROS (Robot Operating System) applications in its multiple virtual machines. This hypervisor kernel is now deployed on all the ground vehicle platforms (LandShark UGV and American Built Car) in the DARPA HACMS (High-Assurance Cyber Military Systems) program.

- They have developed a new set of certified programming methodologies and tools [14] that support programming and composing certified abstraction layers (in C or assembly) and can verify contextual safety, correctness, liveness, and security properties in one unified setting.

- They have developed new semantics and logics [6] for supporting Declarative Decentralized Information Flow Control (DIFC) with declassification. They proposed a new framework which advocate the use of an instrumented semantics for reasoning and the erasure semantics for execution. Their new program logic can be used tor verify security properties for low-level C or assembly programs. They showed that they can prove a new form of non-interference properties even in the presence of declassification. This technology is now being ported into their CertiKOS kernels.

- They have developed new ground-breaking certified resource analysis tools [3, 4] and new logics [25, 26] for verifying safety and liveness of fine-grained shared memory concurrent programs.

- They have made the first comprehensive study [39] that aims to address the architecture deficiencies in all of today's proof assistants. They proposed a new proof-assistant architecture that uses extensible conversion rules and static proof expressions to support effective and

principled proof development. They developed the design, the complete meta theory, and a full compiler of a novel programming language called VeriML [37–39] which realizes the new architecture and also offers a unified platform for coding all kinds of computation on logical terms.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

The ultimate goal of research on certified OS kernels is not just to verify the functional correctness of a particular kernel, but rather to find the best OS design and development methodologies that can be used to build provably reliable, secure, and efficient computer systems in a cost-effective way. The PI and his team at Yale enumerated the following important dimensions of concerns and evaluation metrics which they have used so far to guide their work toward realizing this goal:

- **Support for new kernel design.** Traditional OS kernels use the hardware-enforced "red line" to define a single system call API (Application Programming Interface). A certified OS kernel opens up the design space significantly as it can support multiple certified kernel APIs at different abstraction levels. It is important to support kernel extensions [2, 10] and ring-0 processes [17] so we can experiment and find the best trade-offs.
- **Kernel performance.** Verification should not impose significant overhead on kernel performance. Of course, different kernel designs may imply different performance priorities. An L4-like microkernel [27] would sacrifice portability for faster inter-process communication (IPC) while a Singularity-like kernel [17] would focus on efficient support for type-safe ring-0 processes.
- **Verification of global properties.** A certified kernel is much less interesting if it cannot be used to prove global properties of the complete system built on top of the kernel. Such global properties include not only safety, liveness, and security properties of user-level processes and virtual machines, but also resource usage and availability properties (e.g., to counter denial-of-service attacks).
- **Quality of kernel specification.** A good kernel specification should capture precisely those *contextually observable* behaviors in the kernel implementation [14]. It must support transferring global properties proved at a high abstraction level down to any lower abstraction level.
- **Cost of development and maintenance.** Compositionality is the key to minimize such cost. If the machine model is stable, verification of each kernel module should only need to be done once (to show that it *implements* its deep functional specification [14]). Global properties should be derived from the kernel specification alone.
- **Quality of formal proofs.** They use the term *certified kernels* rather than *verified kernels* to emphasize the importance of third-party machine-checkable proof certificates [35]. Hand-written paper proofs are error-prone [18]. Program verification without machine-checkable proofs has been subject to significant controversy [8].

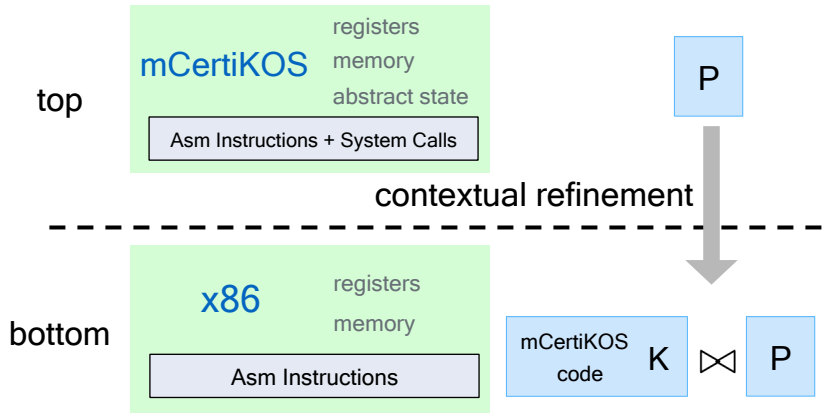


Figure 1: Certified OS kernels: what to prove?

3.1 Overview of the CertiKOS Approach

Their new CertiKOS architecture aims to address all of the above concerns and also tackle all four challenges described in Sec. 2. The CertiKOS architecture leverages the new languages and tools [14] which the PI and his team have developed recently for building certified abstraction layers with deep specifications.

A *certified layer* is a new language-based module construct that consists of a triple (L_1, M, L_2) plus a mechanized proof object showing that the layer implementation M, built on top of the interface L_1 (the *underlay*), is a *contextual refinement* of the desirable interface L_2 above (the *overlay*). A deep specification (e.g., L_2) of a module (e.g., M) captures everything *contextually observable* about running the module over its underlay (e.g., L_1). Once they have built a certified layer M with a deep specification L_2, there is no need to ever look at M again, and any property about M can be proved using L_2 alone. Of course, if the semantics of the underlying abstract machine (for M) changes, the deep specification for M may also have to change.

Under CertiKOS, building a new certified kernel (or experimenting a new design) is just a matter of composing a collection of certified layers, developed in a variant of C (called ClightX) or assembly. The PI and his team [14] have developed a powerful Coq library for supporting *horizontal* and *vertical* composition of certified layers. They have also built a certified compiler (called CompCertX) that can compile certified ClightX layers into certified assembly layers. CertiKOS can thus enjoy the full programming power of an ANSI C variant and also the assembly language to certify any efficient routines required by low-level kernel programming. The layer mechanism allows us to certify most kernel components at higher abstraction levels, even though they all eventually get mapped (or compiled) down to an assembly machine.

In Fig. 1, they use x86 to denote an assembly machine and $[\![\cdot]\!]_{\text{x86}}$ for its whole-machine semantics. Suppose they load such a machine with the mCertiKOS kernel K (in assembly) and

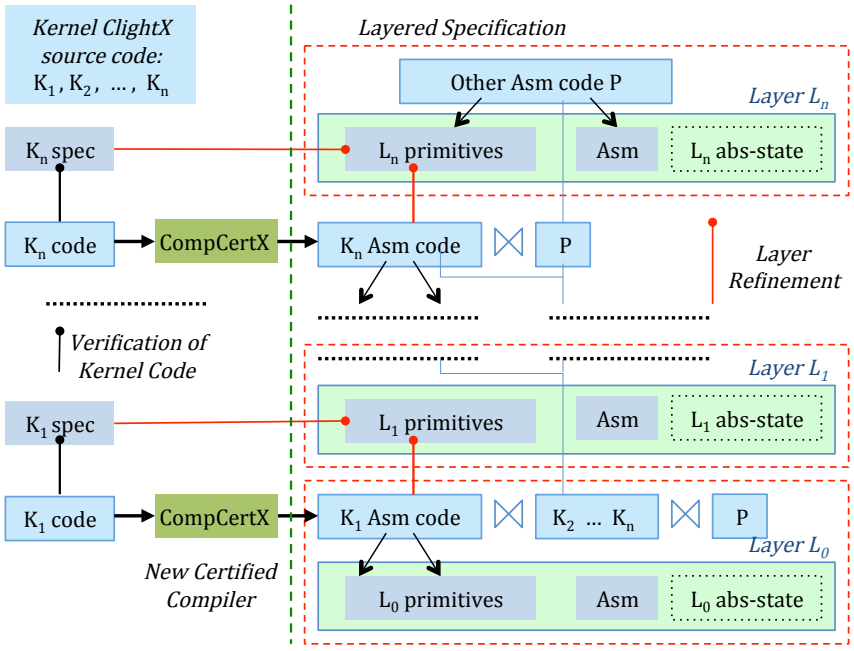


Figure 2: Overview of the CertiKOS architecture

user-level assembly code P; then proving any global property of such a complete system amounts to reasoning about the semantic object $[\![K \bowtie P]\!]_{\text{x86}}$.

Reasoning at such a low level is difficult, so they formalize a new mCertiKOS machine that extends the x86 machine with the deep specification of K. They use $[\![\cdot]\!]_{\text{mCertiKOS}}$ to denote its whole-machine semantics. The contextual refinement property about the mCertiKOS kernel can be stated as $\forall P,\ [\![K \bowtie P]\!]_{\text{x86}} \sqsubseteq [\![P]\!]_{\text{mCertiKOS}}$. Hence any global property proved about $[\![P]\!]_{\text{mCertiKOS}}$ can be transferred to $[\![K \bowtie P]\!]_{\text{x86}}$.

In CertiKOS, they also use contextual refinement to support fine-grained layer decomposition and linking. In Fig. 2, to build a certified kernel K, they decompose it into multiple kernel modules $K_1, ..., K_n$, each sitting at its respective underlay (L_0, ..., L_{n-1}). Each such module (K_i) implements the primitives in its overlay (i.e., L_i) but it can only call the primitives in its underlay (L_{i-1}). Using vertical composition [14], from the contextual refinement $\forall P,\ [\![K_i \bowtie P]\!]_{i-1} \sqsubseteq [\![P]\!]_i$ for each layer (they use $[\![\cdot]\!]_j$ to denote the semantics of the L_j machine), they can deduce $\forall P, [\![K \bowtie P]\!]_0 = [\![K_1 \bowtie K_2 \cdots \bowtie K_n \bowtie P]\!]_0 \sqsubseteq [\![K_2 \cdots \bowtie K_n \bowtie P]\!]_1 \cdots \sqsubseteq [\![K_n \bowtie P]\!]_{n-1} \sqsubseteq [\![P]\!]_n$. If they instantiate L_0 and L_n with the x86 and mCertiKOS layers, they get precisely the contextual refinement property of the mCertiKOS kernel. They can also compose intermediate layers in the same way—this makes it much easier to modify existing (or add new) certified kernel modules.

What have they proved? Using CertiKOS, they have successfully built multiple certified OS kernels. For each such kernel, they have always constructed its deep specification and proved its contextual functional correctness property, so all global properties proved at the specification level can be transferred down to the lowest assembly machine.

From the functional correctness property, they immediately derive that all system calls and traps will always run *safely* and also terminate; and there will be no code injection attacks, no buffer overflows, no null pointer access, no integer overflow, etc. They also proved that there is no stack overflow or memory exhaustion in the kernel using recent techniques developed also by the PI's team *et al* [3, 4]. They have also proved an isolation property between the virtual address spaces of user-level processes. All of these properties were proved using the abstract specification provided at the top layer, and then transferred to the lowest assembly machine via contextual refinement.

Assumptions and limitations Outside their certified mCertiKOS kernel, there are only 163 lines of C (for loading ELF binaries) and 38 lines of assembly code (for handling traps) that are not verified.

The mCertiKOS kernel also relies on a bootloader, whose verification is left for future work. The bottom-most x86 layer of our certified kernels is called *PreInit*, which initializes the drivers, e.g., serial, disk, console, *etc*. Device drivers are not verified because our current machine semantics lacks device models for expressing the corresponding semantics.

Their assembly-level machines do not cover the full x86 instruction sets, so their contextual correctness results only apply to programs in this subset. However, additional instructions and features can be easily added if they have simple or no interaction with our kernel.

The CompCert assembler for converting assembly into machine code is also not verified. They assume the correctness of the Coq proof checker and its code extraction mechanism.

Their current certified kernels assume a runtime environment consisting of a single processor, but extending it to support multicore concurrency is already under way. Their choice of using contextual refinement to compose layers is motivated partly by its close connection [13, 26] with the work on concurrent objects [15, 16].

Like most existing verified kernel efforts, they assume that interrupts are only enabled in user or guest mode. The challenges in handling interrupts and preemption are similar to those for concurrency [11, 12]. They believe that similar approaches can be readily supported in their CertiKOS framework.

Comparison with seL4 The seL4 team [19] focused on verifying a particular microkernel. The designers of the L4-family kernels [9, 27] advocated the *minimality principle*: a concept is tolerated inside the microkernel only if moving it outside the kernel would prevent the implementation of the system's required functionality. This is a reasonable principle but its interpretation of the

"kernel-user" boundary (as the hardware-enforced "red-line") is quite narrow. The PI and his team's new CertiKOS architecture advocates replacing the traditional "red line" with a large number of certified abstraction layers enforced by formal specification and proofs; hardware mechanism (such as address protection) is just one (quick) way of ensuring that a specific process will not violate the invariants required by a particular kernel abstraction layer.

As mentioned in Sec. 2, the seL4 team only proved the *refinement* property but not the *contextual refinement* property, so the global properties (e.g., security [30, 34]) proved at the abstract specification level cannot be transferred to the C-implementation level. The root cause of this problem is their rather simplistic C-level state machine which they used to verify their 7500 lines of C code. This machine is too high level to model several key OS features (e.g, kernel initialization, context switches, address translation, and page-fault handling). Indeed, these features happen to coincide with the unverified C and assembly code in their kernel.

Sewell *et al.* [33] used translation validation to build a refinement proof between the semantics of the verified C source code and the corresponding binary (compiled by GCC). This proof is not as high quality as the rest of the seL4 effort because it was not done in a proof assistant (thus it has no machine-checkable proof) and the translation validator itself still has not been verified.

Even with this work by Sewell *et al.* [33], the previously unverified C code (1200 lines) and assembly code (600 lines) in seL4 still remain unverified. These are actually quite *major* assumptions for a verified kernel because they include the correctness of context switches, kernel initialization, address translation, and linking between verified C and assembly; all of which were considered as major challenge problems by many researchers working in this field [5, 11, 31, 32, 41].

Using CertiKOS, the PI and his team have successfully tackled all of these challenges: context switches, kernel initialization, address translation, and page fault handling are all certified. All kernel components (in C and assembly) are correctly linked together to form a complete system in an assembly machine and all our proofs are machine-checkable in Coq.

Much of the implementation complexity of the seL4 kernel lies on its support of capability-based access control. Capabilities are important in seL4 as they are used to prevent unwanted interference between different kernel components. However, they significantly increase the complexity of the seL4 kernel. In contrast, the CertiKOS-family kernels the PI and his team have built so far rely on the CompCert memory model [23] to enforce isolation and prove contextual refinement.

3.2 Methods and Procedures: Defining Abstraction Layers

Contextual refinement provides an elegant formalism for decomposing the verification of a complex kernel into a large number of tractable tasks: the PI and his team define a series of logical abstraction layers, which serve as increasingly higher-level specifications for an increasing portion of the kernel code. They design these abstract layers in a way such that complex interdependent kernel

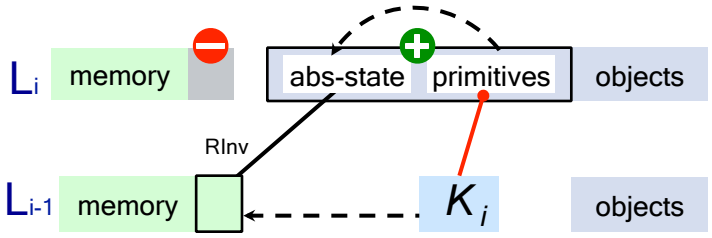


Figure 3: Introducing a new layer object

components are untangled and converted into a well-organized kernel-object stack with clean specification.

Their framework specifies an abstraction layer using five components: a collection of objects, a memory model, an invariant which the memory and objects satisfy at any point of the execution, an initialization flag, and an initialization primitive. These five components define a logical view of a subset of the kernel code and extend our language with an abstract specification of that code. On top of this logical view, more code is introduced and verified.

Layer objects The layer objects are logical abstractions of kernel modules. In Fig. 3, each layer object provides a set of abstract states (which are abstractions of the module's private memory) and a set of primitives (which are abstractions of the module's interface specified in terms of the abstract states). Consecutive layers may reuse some of the same objects, introduce new layer objects by verifying additional code, or hide some low-level objects which are used to implement new objects but need not be exposed to higher layers. Hiding unnecessary objects facilitates invariant proofs since they can often use stronger invariants at higher layers that would otherwise be violated by low-level objects.

For example, thread queues are implemented as doubly-linked lists in mCertiKOS, and the concrete implementations of the functions that manipulate queues (*enqueue* and *dequeue*) directly manipulate these doubly-linked lists in memory. On the other hand, in our abstract queue layer object, a queue is just a simple list of thread identifiers, and the *enqueue* and *dequeue* primitives are specified directly over the abstract lists. The contextual refinement relation between the two layers (one with concrete implementation and the other with the abstract layer object) ensures that any kernel/user context code (e.g., the scheduler) running on top of the more abstract layer retains an equivalent behavior when it is running on top of the layer with corresponding concrete implementation.

As shown in Fig. 3, to establish the contextual refinement relation between concrete memory and abstract state, they use CompCert memory permissions [24] at the higher layer to prevent the context code from accessing the module's private memory. Note that these permissions do *not* correspond to a physical protection mechanism, but instead are entirely logical: they ensure that the

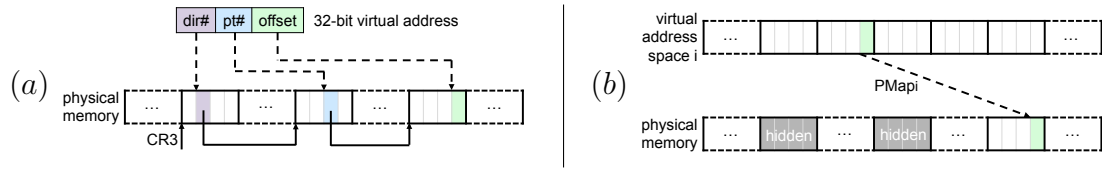


Figure 4: (a) Machine memory model; (b) Abstract memory model

higher-level abstract machine gets stuck whenever it executes code that directly accesses this private memory. By proving our kernel is safe (it does not get stuck), they guarantee that this situation will not happen.

Memory models OS kernels must manage limited physical memory and provide contiguous address spaces for high-level kernel modules and user programs. Because much of the code assumes that the memory management sets up the virtual address space properly, initialization has been a sticking point in previous verification efforts [20, 41], in which the virtual address space setup is either not verified, or verified separately as an external lemma. They address this challenge by making the memory model explicit in our abstraction layers.

Because they use CompCertX [14] along with its formalization of the semantics of C and assembly, our notion of memory is based on the CompCert memory model [24]. CompCert employs a unified model to encode different views of memory. The memory is split into a number of disjoint blocks and a pointer is represented by a pair (b, o), where b is a block identifier and o is an offset within block b. Each offset within a block is associated with a permission specifying the memory operations that can be performed at that location. A program which attempts to perform a prohibited operation will get stuck. The compiler's correctness theorem guarantees that the target program will only get stuck if the source does; thus the compiler will never introduce invalid memory operations into a correct program.

In CompCert, this unified memory model built around blocks and permissions is used to encode different views of the memory. For instance, at the C level each variable is assigned its own memory block, so that the semantics of CompCert C reflect the C standard by invalidating pointer arithmetic across variable boundaries. On the other hand, in the emitted assembly code, a function's local variables are merged into a single "stack frame" memory block. CompCert's simulation proof has to keep track of the correspondence between these two views of the memory, but the fact that the semantics of the source and target languages are expressed in terms of a unified memory framework tremendously simplifies the compiler's verification. At the assembly level, this model is still slightly more abstract than the hardware, yet it is sophisticated enough that CompCert's stack layout pass, for instance, can be properly verified.

They follow a similar approach, and extend the semantics of CompCert assembly so that the

CompCert memory model can be equipped with notions of page fault and address translation. A distinguished block is used to represent the entire address space. The memory model of a layer L specifies how memory loads and stores are carried out in terms of the system description at that level of abstraction. The machine memory model, and those implemented by the physical and virtual memory management components, organize memory in terms of various units (byte, page, address space), and provide different addressing modes and protection mechanisms. Because our kernel code is compiled using CompCertX, its own stack frames and static data have to be modeled as independent blocks. However, as explained in Sec. 4, we prove that user programs can never access the kernel portion of the address space. They also use an external tool [3] to prove that the stack usage of our compiled kernel is bounded such that stack overflows cannot occur: the computed bound is much less than the dedicated 4K bytes we use for kernel stacks.

Integrating the various views of the memory into our layered approach allows us to reason about memory accesses in the same way that we reason about other kernel services: as long as the low-level machine memory model, as configured by our kernel code, contextually refines a more abstract memory model, any code we can write and reason about in terms of the latter can be shown to have an equivalent behavior when run on top of the former. As shown in Fig. 4(a), the *machine memory model* is an unstructured CompCert memory block, which is consistent with the hardware view of the physical memory. Accesses to this memory block are modeled in a way that mirrors the operation of the paging hardware. By contrast, in the top-level memory model (which we call the *abstract memory model*), address translation cannot be disabled; memory accessors operate on the basis of the high-level, abstract descriptions of address spaces rather than concrete page directories and page tables stored in the memory itself (see Fig. 4(b)).

Layer invariant Each abstraction layer specifies a predicate on the memory and layer objects' abstract states. This invariant is satisfied by the initial state and preserved by memory accessors and the layer objects' primitives. It therefore holds in all client contexts, at any point of execution.

In previous verification efforts, proving invariants has typically been challenging. For example, in seL4, the thread queues are implemented as doubly-linked lists with the following invariant:

Invariant 1. *All back links in thread queues point to appropriate nodes and all elements point to thread control blocks.*

Proving this invariant is difficult for several reasons. As stated in [20]:

> Invariants are expensive because they need to be proved not only locally, but for the whole kernel — we have to show that no other pointer manipulation in the kernel accidentally destroys the list or its properties. [...] The treatment of globals becomes especially difficult if the invariants are temporarily violated. For example, adding a new node to a doubly-linked list temporarily violates invariants that the list is well formed.

However, in our layered approach, global variables and the code that manipulates them are abstracted as layer objects. The remaining kernel code cannot access the abstracted variables directly, since they are hidden using CompCert memory permissions. Moreover, the abstract primitives are atomic, hence there is no longer a point in the execution at which the invariants have to be temporarily violated. Finally, some complex invariants are implied by the correspondence with our abstract representations. For instance, in our setting, Inv. 1 naturally follows from the contextual refinement between concrete thread queues and abstract "thread list" objects.

Initialization flag and primitive Each layer has exactly one initialization primitive, which can be viewed as a special layer object together with the initialization flag. This logical initialization flag is *false* in the initial state and is set to *true* by the initialization primitive. Most of the invariants and specifications of non-initialization primitives require as a precondition that the initialization flag is *true*. This guarantees that the initialization primitive is the first primitive that is executed.

3.3 Methods and Procedures: Introducing Abstraction Layers

Introducing new layers is a way to organize code and lift the abstraction level. In most cases, this does not require modifying the implementation. In this section, the PI and his team discuss some of the principles they used when drawing the boundaries of their kernel's abstraction layers.

Principle 1: Introduce layers to reflect dependencies between kernel modules One purpose of layers is to enforce code isolation and abstraction. When a module M depends on another module N, abstraction layers should be organized in such a way that M can be reasoned about in terms of an abstracted version of N.

For example, since the virtual memory management code relies on physical memory management, the code which performs allocation and deallocation of physical pages in terms of allocation tables is first abstracted into a layer object. This object provides the primitives *palloc* and *pfree* and defines their abstract specifications. Then functions such as *pt_insert* and *pt_rmv*, which manipulate page mappings at the virtual memory management level, can be verified with a more abstract view of the allocation table, without worrying about its concrete memory representation and code implementation. On the other hand, if two kernel modules mutually depend on each other, they have to be introduced within a single layer.

Principle 2: Introduce a layer when the memory model changes In the *machine memory model*, when paging is enabled, each memory access is accompanied by a two level page table walk starting from the address stored in the *CR3* register, shown in Fig. 4(a). Switches of page tables are performed by storing the top address of the other page table structure into *CR3*. In the *abstract*

memory model, we associate with each process a logical partial map from a virtual address to a pair of physical address and permission. The address translations are performed using the logical mappings of the currently-running process, shown in Fig. 4(b). With this high level memory model, some complex properties like memory isolation can be proved more easily.

mCertiKOS uses an additional intermediate memory model. The mCertiKOS-hyp extension presented in Sec. 4 uses yet another, virtualization-related model. They introduce a new layer whenever we switch from one memory model to another and establish the contextual refinement between them.

Principle 3: Introduce a layer when a stronger invariant needs to be proved After paging is enabled, both kernel modules and user processes run in a virtual address space. To ensure the correctness of these kernel modules and user processes on top of virtual memory management, we require the following invariants to hold:

Invariant 2. *1) paging is enabled only after the initialization of virtual memory management; 2) the memory regions that store kernel-specific data must have the kernel-only permission in all page maps; 3) the page map used by the kernel is an identity map 4) the non-shared parts of user processes' memory are isolated.*

Inv. 2 no longer holds if the privileged primitive that sets the *CR3* register is present in the layer, as the unknown context code may write an invalid address into *CR3* using the provided primitive. To solve this issue, another layer is introduced with a wrapper function that takes the process id as argument, instead of an actual address. Then the function sets *CR3* to the starting address of the predefined corresponding process's page table structure. The primitive that directly sets the *CR3* register is hidden from the new layer, and the invariants are introduced in the new layer. This is one of the rare cases where performance overhead is introduced (one extra function call due to the wrapper). It should be possible to use CompCertX's function-inlining optimization to remove this overhead (this is left as future work).

Principle 4: Introduce a layer to facilitate initialization proofs Recall that each layer contains one initialization primitive. This primitive can be passed through from the layer below, or a new one can be defined which extends that of the layer below so as to initialize the new layer's data. When a new layer object is introduced, we can create a new layer to initialize its abstract data to an appropriate state. In the context of an operating system kernel, initialization functions are relatively complex. Introducing an extra layer allows us to avoid directly reasoning over the concrete memory. With this new layer, an initialization function is verified using a more abstract specification.

4 RESULTS AND DISCUSSION

4.1 Certifying the mCertiKOS Kernel

In this section, the PI and his team describe the main parts of the certification of mCertiKOS. The mCertiKOS kernel is divided into four main components (see Fig. 5) which consist of multiple layers: the pre-initialization module (1 layer), the memory management (14 layers), the process management (14 layers), and the trap handler (4 layers). The pre-initialization module contains the bottom layer that corresponds to the physical machine and trap handler contains the top layer provides system calls and serves as a specification of the whole kernel. Their main theorem states that context code that is understood in terms of the topmost abstraction layer has an equivalent behavior when run along with the kernel on the bottom-most layer.

The overall structure of the layered certification is shown in Fig. 5. Each row in the diagram describes a layer. It consists of the name of the layer (on the very left) followed by the initialization primitive (green background), and the memory model used by the layer (blue background). The rest of the row describes layer objects, each in their own bordered rectangle. Normal white-filled objects are used to implement new layers, while those filled with gray are hidden from higher layers. Some objects span across multiple rows and are colored purple, meaning that they are horizontally composed to implement higher layers. The objects with different subscripts indicate different abstract view over the same data.

Pre-initialization module The pre-initialization module only contains the bottom-most layer *PreInit*. It is used to model the x86 hardware and axiomatizes the hardware behaviors that are necessary to obtain end-to-end behaviors across the kernel and the user space. These behaviors include page table walk upon memory load when paging is turned on, saving and restoring part of the trap frame in the case of interrupts, and switching the stack pointer in the case of ring switch.

The *x86* object is the only layer object in the PreInit layer. It extends the CompCert assembly semantics to model the low-level features of the machine. Its abstract state consists of control registers, a physical memory map *MM*, and a kernel mode flag *ikern*. Its primitives consist of getter-setter functions for control registers and *MM*, and a function models the transition between user and kernel mode.

The state component *MM* is the abstraction of the E820 memory map provided by the bootloader. The control registers, such as *CR0*, *CR2*, and *CR3*, are used to model the behavior of the processor's memory management unit (MMU). When paging is enabled (as indicated by *CR0*), memory accesses made by both the kernel and the user programs are translated using the page map pointed to by *CR3* in the *machine memory model*. When a page fault occurs, the corresponding information is stored in *CR2* and the page fault handler is invoked. The logical flag *ikern* indicates whether the processor is currently in the kernel or user mode. Some privileged memory regions (e.g., allocation table, page

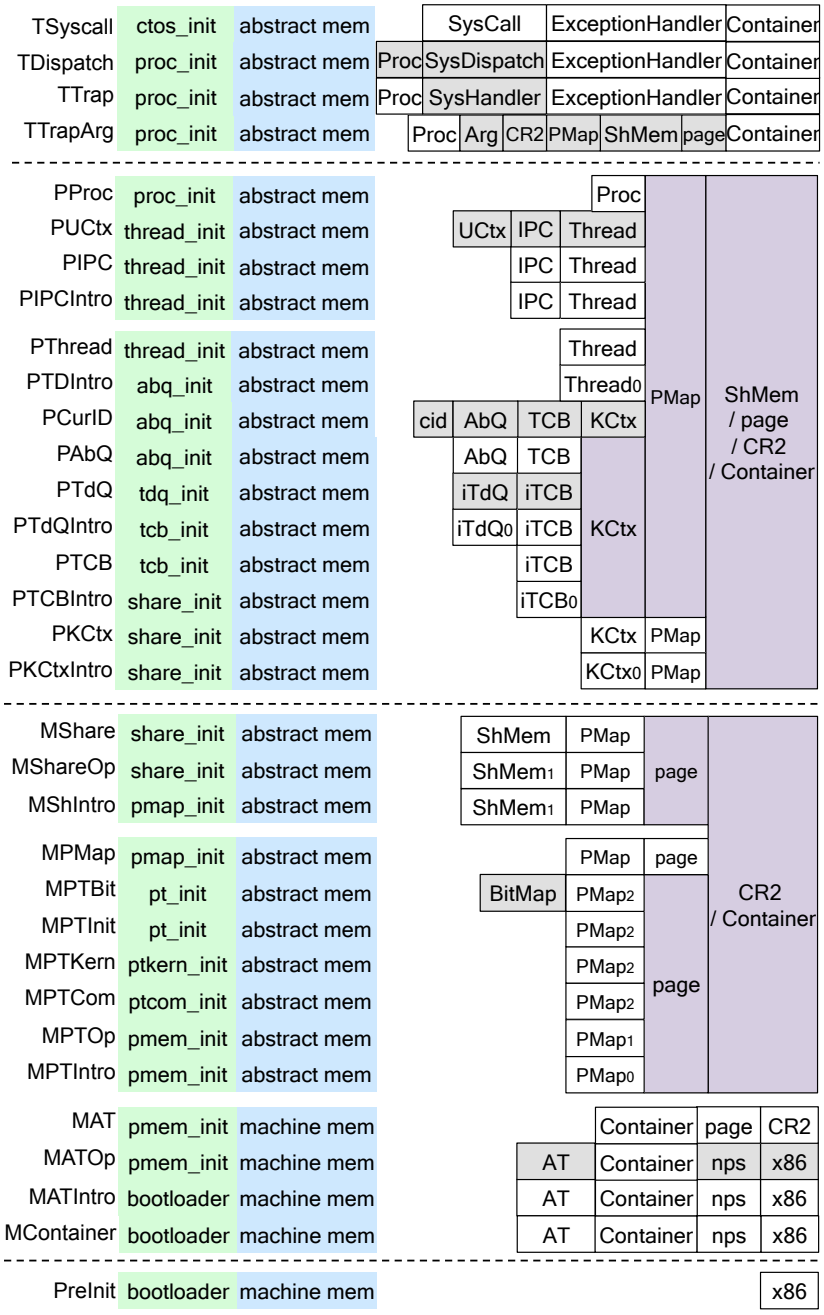


Figure 5: Layers of mCertiKOS

map) and instructions (e.g., modifying control registers) are only available in the kernel mode.

Memory management The memory management of mCertiKOS consists of the physical memory management (4 layers), virtual memory management (7 layers) and shared memory management (3 layers).

Based on the pre-initialization layer and the *machine memory model*, the physical memory management abstracts the physical page allocation table into *page* objects. To better reason about access control and isolation in the case of the dynamic resource allocation, each physical page object maintains a *logical* state containing ownership information, and the page is only allowed to be accessed by its owners.

On top of physical memory management, the virtual memory management provides consecutive virtual address spaces. They proved not only that the primitives of virtual memory management manipulate the address space correctly, but also that the initialization procedure sets up the two-level page maps properly in terms of hardware address translation. The Inv. 2 they have proved guarantees that it is safe to run both the kernel and user programs in the virtual address space when paging is enabled.

The shared memory management provides a protocol to share physical pages among different user processes. It provides an infrastructure to map a physical page into multiple processes' page maps in different address spaces. Their ownership mechanism ensures that the page can only be freed once all processes release ownership.

Enforcing memory quotas Another function of the physical memory management is to dynamically track and bound the memory usage (in terms of number of dynamically-allocated pages) of processes based on their id.

In mCertiKOS, they consider every unique integer (up to some predefined maximum, currently 2^{18}) to represent a different agent or principal. They refer to this integer as the agent's id, and they use it for all layer objects owned by that agent.

The MContainer layer introduces a notion of container, inspired by container objects in the HiStar operating system [42]. Whenever a new agent (id) is created in mCertiKOS, a container is created for the agent that dynamically keeps track of its memory usage. An agent's usage may increase for a few reasons, including a direct request for dynamically-allocated resources, or a successfully-handled page fault. Each container object is initialized with some maximum *quota*; any attempt for an agent to increase its usage beyond this quota will be denied by the kernel. Furthermore, the kernel maintains a mapping of ids to containers using a hierarchical tree structure. Whenever an agent's process makes a request to spawn a new process, the new container is added as a child to the requesting agent's container, and the new container's quota is taken from the requester's.

With this notion of container, they are able to prove a theorem about reliability of dynamic memory allocation: agents' requests for additional resources will always be fulfilled as long as their quota is not exceeded. Furthermore, from the viewpoint of information-flow security, resource quotas close the potential for two different processes to communicate via allocation requests. Hence quota enforcement provides an additional level of security for mCertiKOS. They plan to extend the concept of containers to other types of resources in the future. For example, they could maintain a time-slice quota for each agent. This would provide a foundation for reasoning about liveness properties for processes and security breaches via timing channels.

Process management Process management depends on virtual address spaces and introduces the *thread* and *proc* objects as the abstractions of threads and processes, respectively. One interesting aspect of the process management component is the context switch function. This assembly function saves the register set of the current thread and restores the register set from the kernel context of another thread. Since the instruction pointer register (*EIP*) and stack pointer register (*ESP*) are saved and restored in this procedure, they can show that this function reflects the C-level behavior and restores the continuation of a thread's execution. Even though this kernel context switch function is verified at assembly level, they prove that it will not violate the convention of ClightX execution. This enables us to link it with other code that is verified at C-level and compiled by CompCertX.

In the process management component, they have also implemented and verified a single-copy synchronous inter-process communication (IPC) protocol. Additionally, they have verified an asynchronous zero-copy IPC implementation that is built on top of their shared memory infrastructure.

Trap module The trap module specifies the behaviors of exception handlers and mCertiKOS system calls. In mCertiKOS, exception handlers are registered in a table of first-class code pointers. When an exception triggers (via interrupt), the kernel consults this table and invokes the corresponding exception handler. For example, a page fault at the user level traps into the kernel. The page fault handler then reserves a page for PFLA (if necessary) and returns to the user level. The verification of the page fault handler depends on layer objects introduced at different abstraction levels (see Fig. 6). Therefore, the behavior of the page fault handler is interpreted by the concrete first-class code pointer until all the dependent layer objects are introduced. Then the handler code is verified and the behavior is interpreted using its abstract atomic specification.

To further simplify the reasoning about user code, they have implemented and verified the user level system call libraries directly in the user space. Since their machine semantics models hardware behaviors like paging and ring switch, the specifications of user system call libraries closely corresponds to the real execution model in the actual hardware. With this atomic system call semantics in the user level, the user code can be proved much more easily.

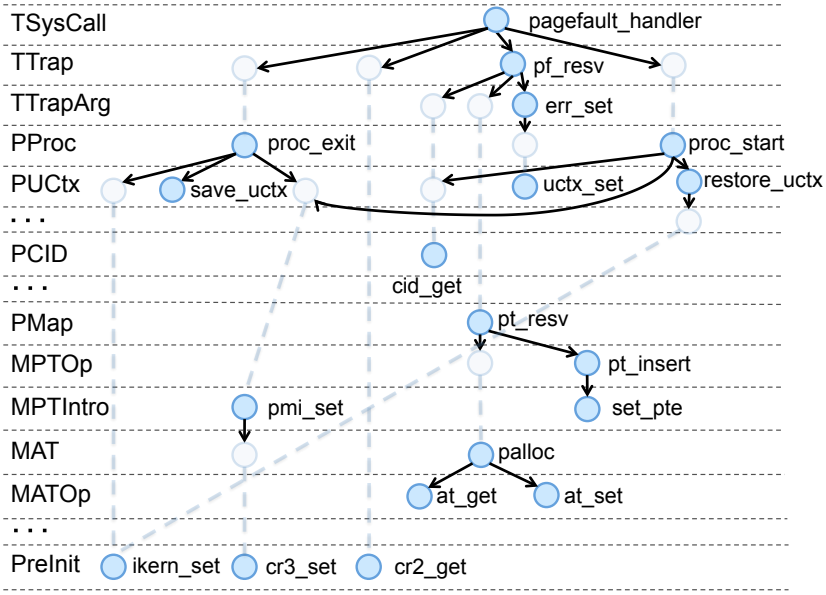


Figure 6: Call graph of page fault handler

4.2 Extensions and Adaptation

One primary advantage of the PI and his team's new extensible architecture is that it makes certified kernel extension and reasoning much easier and more principled. In this section, they describe three alternative mCertiKOS kernels that they created through relatively minor changes to the base kernel. They then present a specific example of global reasoning over the mCertiKOS kernel — a simple notion of address space isolation that will serve as a starting point for a full-fledged security proof in the future.

mCertiKOS-hyp: supporting virtualization They also augmented mCertiKOS to support the two hardware-assisted virtualization technologies Intel VT-x and AMD SVM, and built a certified hypervisor mCertiKOS-hyp.

Fig. 7 shows the 7 layers of the virtual machine management of mCertiKOS-hyp on the Intel platform. *VMInfo* is the layer object that axiomatizes some of the hardware specific features needed for the virtualization support. Since it is orthogonal to memory and process management, the *VMInfo* object can be horizontally composed with the layers below *PProc* in mCertiKOS. On top of this extended *PProc* layer, the virtual machine management extends the *abstract memory model* with the notions of Extended Page Table (EPT), the virtual machine control structure (VMCS), and the virtual machine extension meta data (VMX), which are abstracted into corresponding layer objects. These objects are again orthogonal to the trap module above and can be horizontally composed to export related system calls with minimal cost.

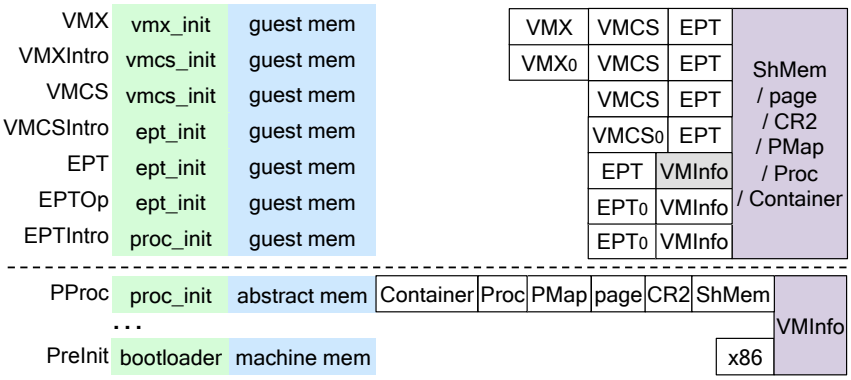


Figure 7: Layers of virtual machine management

mCertiKOS-rz: supporting ring 0 processes Thanks to the contextual refinement relation they have proved for mCertiKOS, one can certify user programs using their formal specifications of system calls. This gives end-to-end proofs on the behaviors of user programs when they run on mCertiKOS. Furthermore, once certified, these processes can safely run in the privileged ring 0 mode. They extended mCertiKOS into mCertiKOS-rz by adding support for spawning "in-kernel processes" that run in the privileged ring 0 mode. Ring 0 processes get much better system call performance by directly calling kernel functions and avoiding ring switch and interrupt processing.

To introduce ring 0 processes to mCertiKOS, they added a single layer on top of the existing process management module: Spawning a ring 0 process sets the initial *ESP* register to a preallocated memory region and then spawns a proper kernel thread. The memory region must be verifiably sufficient for the entire execution of the process.

mCertiKOS-emb: embedded systems The mCertiKOS-emb kernel is intended for embedded settings. To develop this kernel they started with mCertiKOS-rz and removed the virtual machine management, the virtual memory management, and some of the process management layers that are related to user contexts and user process management. Thus mCertiKOS-emb only supports ring 0 processes which run directly inside the physical kernel address space instead of the user-level paged virtual address space.

Removing plug-ins or layers does not take much effort. They only need to alter the contextual refinement proof at the boundary so they can glue them back together.

Isolation in mCertiKOS They have begun exploring the verification of a global security property on top of mCertiKOS. As a starting point, they proved a basic notion of isolation between user-level processes running in different virtual address spaces. This isolation property is composed of two theorems: one regarding integrity (write protection), and another regarding confidentiality (read

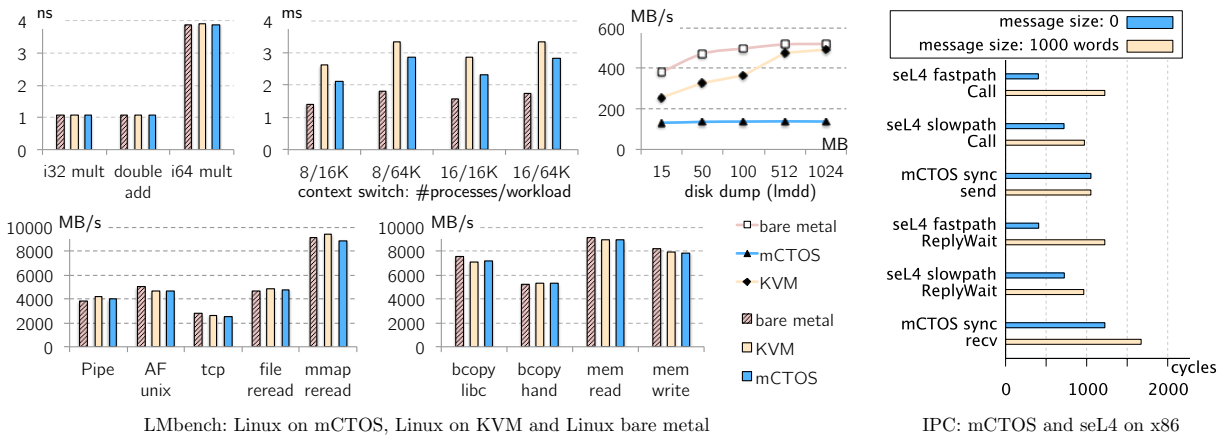


Figure 8: Performance evaluation with micro benchmarks.

protection, or noninterference). The statements of these two theorems are as follows: suppose the top layer abstract machine takes one step, changing the machine state from S to S', and let p be the id of the currently-running process (which can be found in S).

Integrity: If the value at some non-kernel memory location l differs between S and S', then l belongs to a page that is mapped in the virtual address space of p.

Confidentiality: If the step taken is not a primitive call to an IPC syscall (send, recv, etc.), then the values of memory in any address space other than p's cannot have an effect on the result of the step. In other words, if they altered S by changing data in a different process's address space, the step would still have the same effect on p's address space.

In the future, they plan to provide a more detailed security policy by describing what can happen to confidentiality when IPC is used. This description will be expressed in terms of propagation of security labels on the IPC data. Note, however, that their framework allows for security labels to be specified at a purely logical level — there is no need for concrete representation and manipulation of labels at run time.

Noninterference properties are generally not preserved across refinement due to nondeterminism. It may therefore seem that the aforementioned *confidentiality* holds only at the topmost layer, but not at lower layers. It turns out, however, that their notion of deep specification is strong enough to preserve noninterference. Essentially, to give a deep specification to a nondeterministic semantics, they must first externalize the source of nondeterminism (e.g., into an oracle). The noninterference property then becomes parameterized over this source of nondeterminism, which allows the parameterized property to be preserved across refinement. This relationship between deep specification, noninterference, and refinement will be explored comprehensively in future work.

4.3 Performance Evaluation and Proof Effort

The PI and his team have also analyzed the performance of the mCertiKOS-hyp hypervisor kernel with a thorough experimental benchmark evaluation. Furthermore, an extended version of mCertiKOS-hyp was deployed in a practical system that is used in the context of another related research project funded by the DARPA HACMS program. Their experiments with benchmarks confirm the observations made during deployment: the performance overhead of mCertiKOS-hyp is moderate. They are convinced that it is practical to use their verification framework to produce competitive real-world kernels with acceptable effort.

Performance evaluation They used a number of micro and macro benchmarks to measure the overhead of mCertiKOS-hyp and to compare mCertiKOS-hyp to existing systems such as KVM and seL4. All experiments have been performed on an Intel Core i7-2600 S with 8 MB L3 cache, 16 GB memory, and a 120 GB Intel 520 SSD. Since the power control code has not been verified, they disabled the turbo boost and power management features of the hardware during experiments.

A comparison of the performance of seL4 and mCertiKOS-hyp is not straightforward since the mCertiKOS kernels run on x86 platforms but the verified seL4 runs on ARMv6 and ARNv7 hardware. Moreover, the verified version of seL4 does not have virtualization support and cannot boot Linux. As a result, they do not compare hypervisor performance but instead focus on a comparison of the IPC performance of mCertiKOS-hyp and an unverified x86 version of seL4.

IPC Performance They compared IPC in mCertiKOS-hyp and the (unverified) x86 version of seL4. They used seL4's IPC benchmark sel4bench-manifest[1] with processes in different address spaces and with identical scheduler priorities, both in *slowpath* and *fastpath* configurations. To run this benchmark on mCertiKOS-hyp, they replaced seL4's *Call* and *ReplyWait* operations with mCertiKOS-hyp synchronous *send* and *receive* operations. Fig. 8 (on the right) contains a compilation of their results. It shows the average number of clock cycles needed for the operations for message sizes 0 and 1000.

Because seL4 follows the microkernel design philosophy, its IPC performance is critical. IPC implementations in seL4 are highly optimized, and heavily tailored to specific hardware platforms. While this degree of optimization gives seL4 an advantage in IPC intensive systems, they currently do not see the need to improve IPC performance in mCertiKOS-hyp for application scenarios of the kernel that they have in mind.

Hypervisor Performance To evaluate mCertiKOS-hyp as a hypervisor, they measured the performance of micro and macro benchmarks on Ubuntu 12.04.2 LTS running as a guest.

[1] https://github.com/smaccm/sel4bench-manifest

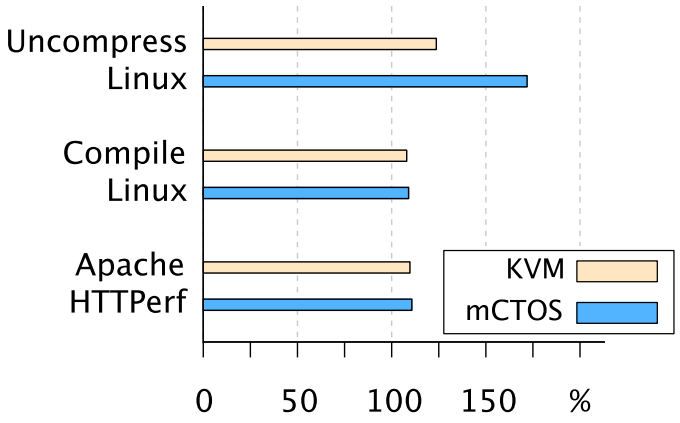


Figure 9: Normalized macro benchmarks: Linux on KVM and mCTOS, baseline is Linux on bare metal

Fig. 9 contains a compilation of standard macro benchmarks: unpacking of the Linux 4.0-rc4 kernel, compilation of the Linux 4.0-rc4 kernel, and Apache HTTPerf. They ran the benchmarks on Linux as guest in KVM and mCertiKOS-hyp, as well as on bare metal. In Fig. 9 they normalized the run times of the benchmarks using the bare metal performance as a baseline (100%). The overhead of mCertiKOS-hyp is moderate and comparable to KVM. They attribute the larger overhead for decompression to their unverified SSD driver that still contains performance bugs (compare *disk dump* in Fig. 8).

Fig. 8 (on the left) shows a compilation of micro benchmarks from the LMbench benchmark suite [29]. They measure the performance of the file system, some local communication systems, virtual memory, context switch and, for sanity checking, basic arithmetic operations. On the x-axes of the plots are the names of the respective LMbench benchmarks. The y-axes of the two plots at the top left show the run time in nanoseconds and microseconds, respectively. The other three y-axes show the throughput in MB/s.

In many cases, the performance of mCertiKOS-hyp is in between bare metal and KVM (Kernel Virtual Machine). However, there are still some rough edges in the results that they mostly attribute to performance problems with their unverified SSD driver. This is indicated for instance by the disk dump benchmark in which the transfer rate seems to remain constant as the data size increases. They are currently investigating the issue.

The virtualization drivers in mCertiKOS-hyp are running in a user process in the ring 3 mode. This approach makes the kernel smaller and makes it possible to use an unverified driver. The downside of this approach is that each VM entry and exit causes an additional ring switch, and VM-related information must be copied to the user driver process in order for it to process the exit. Therefore, it may have an impact on performance, especially for those guest programs that frequently cause VM exits, such as web servers, which generate frequent network-related external interrupts. Another approach is to verify the drivers and run them inside a kernel module, e.g., in a

ring 0 process.

Proof effort The PI and his team completed the verification of mCertiKOS-hyp in less than 18 person months (pm). The layer design and verification took about 3 pm for the physical memory management (4 layers), 3.5 pm for the virtual memory module (7 layers), 1 pm for the shared memory infrastructure (3 layers), 3.5 pm for the thread management (10 layers), 1 pm for the process management (4 layers), 1.5 pm for the trap handler module (4 layers), 1.5 pm for the AMD SVM virtualization (9 layers), and 2 pm for the Intel VT-x virtualization support (7 layers). In total the verified mCertiKOS-hyp kernel consists of 5500 lines of C and x86 assembly code.

The verification effort roughly falls into three categories: layer design with specification and invariants, refinement proofs between the layers, and verification of C and assembly code with respect to the specifications. The time needed for each of the categories depends largely on the layer. For instance, at the boundary of physical and virtual memory management (*MPTIntro*), almost all effort is in the refinement proof, due to the proof for the refinement between two completely different memory models. More effort went into the refinement proof when they introduced the Intel *virtual machine memory model*, where they proved the refinement between the concrete four level extended page table structure in memory and the abstract mapping from the guest addresses to the host addresses. In contrast, for the layer *MATOp*, which initializes physical memory allocation, most of the time was spent on verifying the non-trivial nested loops present in the C code, while the refinement proofs were derived automatically.

The proofs were facilitated by automation tools for C code, layer design patterns, and tactics libraries developed in recent years [14]. These tools have greatly reduced the amount of work needed to verify extensions of the kernel.

4.4 Other Important Results

In addition to developing new cutting-edge technologies for building certified OS kernels, the PI and his team have also obtained the following important results. We annotate each technology with a publication venue where the main result is first published. Here, POPL refers to "ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages;" PLDI refers to "ACM SIGPLAN Conference on Programming Language Design and Implementation;" ESOP refers to "European Symposium on Programming;" APLAS refers to "Asian Symposium on Programming Languages and Systems;" CPP refers to "International Conference on Certified Programs and Proofs;" LICS refers to "IEEE International Conference on Logic in Computer Science;" POST refers to "International Conference on Principles of Security and Trust;" CONCUR refers to "International Conference on Concurrency Theory." All papers referenced here are attached in the Appendix.

Deep specifications and certified abstraction layers (POPL'15) Modern computer systems consist of a multitude of abstraction layers (e.g., OS kernels, hypervisors, device drivers, network protocols), each of which defines an interface that hides the implementation details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Despite their obvious importance, abstraction layers have mostly been treated as a system concept; they have almost never been formally specified or verified. This makes it difficult to establish strong correctness properties, and to scale program verification across multiple layers.

In this work, the PI and his team present a novel language-based account of abstraction layers and show that they correspond to a strong form of abstraction over a particularly rich class of specifications which they call *deep specifications*. Just as *data abstraction* in typed functional languages leads to the important *representation independence* property, abstraction over deep specification is characterized by an important *implementation independence* property: any two implementations of the same deep specification must have *contextually equivalent* behaviors. They present a new layer calculus showing how to formally specify, program, verify, and compose abstraction layers. They show how to instantiate the layer calculus in realistic programming languages such as C and assembly, and how to adapt the CompCert verified compiler to compile certified C layers such that they can be linked with assembly layers. Using these new languages and tools, they have successfully developed multiple certified OS kernels in the Coq proof assistant.

Compositional Certified Resource Bounds (PLDI'15) In this work, the PI and his team developed a new approach for automatically deriving worst-case resource bounds for C programs. The described technique combines ideas from amortized analysis and abstract interpretation in a unified framework to address four challenges for state-of-the-art techniques: compositionality, user interaction, generation of proof certificates, and scalability. *Compositionality* is achieved by incorporating the potential method of amortized analysis. It enables the derivation of global whole-program bounds with local derivation rules by naturally tracking size changes of variables in sequenced loops and function calls. The resource consumption of functions is described abstractly and a function call can be analyzed without access to the function body. *User interaction* is supported with a new mechanism that clearly separates qualitative and quantitative verification. A user can guide the analysis to derive complex non-linear bounds by using auxiliary variables and assertions. The assertions are separately proved using established qualitative techniques such as abstract interpretation or Hoare logic. *Proof certificates* are automatically generated from the local derivation rules. A soundness proof of the derivation system with respect to a formal cost semantics guarantees the validity of the certificates. *Scalability* is attained by an efficient reduction of bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. The analysis framework is implemented in the publicly-available tool C4B. An experimental evaluation demonstrates the advantages of the new technique with a comparison of C4B with existing tools on challenging micro benchmarks and the analysis of more than 2900 lines of C code from the cBench

benchmark suite.

Automatic Static Cost Analysis for Parallel Programs (ESOP'15) Static analysis of the evaluation cost of programs is an extensively studied problem that has many important applications. However, most automatic methods for static cost analysis are limited to sequential evaluation while programs are increasingly evaluated on modern multicore and multiprocessor hardware. This work introduces the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. The analysis is performed by a novel type system for amortized resource analysis. The main innovation is a technique that separates the reasoning about sizes of data structures and evaluation cost within the same framework. The cost semantics of parallel programs is based on call-by-value evaluation and the standard cost measures *work* and *depth*. A soundness proof of the type system establishes the correctness of the derived cost bounds with respect to the cost semantics. The derived bounds are multivariate resource polynomials which depend on the sizes of the arguments of a function. Type inference can be reduced to linear programming and is fully automatic. A prototype implementation of the analysis system has been developed to experimentally evaluate the effectiveness of the approach. The experiments show that the analysis infers bounds for realistic example programs such as quick sort for lists of lists, matrix multiplication, and an implementation of sets with lists. The derived bounds are often asymptotically tight and the constant factors are close to the optimal ones.

A Compositional Semantics for Verified Separate Compilation and Linking (CPP'15) Recent ground-breaking efforts such as CompCert have made a convincing case that mechanized verification of the compiler correctness for realistic C programs is both viable and practical. Unfortunately, existing verified compilers can only handle whole programs—this severely limits their applicability and prevents the linking of verified C programs with verified external libraries. In this work, the PI and his team present a novel compositional semantics for reasoning about open modules and for supporting verified separate compilation and linking. More specifically, they replace external function calls with explicit events in the behavioral semantics. They then develop a verified linking operator that makes lazy substitutions on (potentially reacting) behaviors by replacing each external function call event with a behavior simulating the requested function. Finally, they show how our new semantics can be applied to build a refinement infrastructure that supports both vertical composition and horizontal composition.

Compositional Verification of Termination-Preserving Refinement of Concurrent Programs (LICS'14) Many verification problems can be reduced to refinement verification. However, existing work on verifying refinement of concurrent programs either fails to prove the preservation of termination, allowing a diverging program to trivially refine any programs, or is difficult to apply in compositional thread-local reasoning. In this work, the PI and his colleague at USTC first propose a new simulation technique, which establishes termination-preserving refinement and is a

congruence with respect to parallel composition. Then they give a proof theory for the simulation, which is the first Hoare-style concurrent program logic supporting termination-preserving refinement proofs. They show two key applications of our logic, i.e., verifying linearizability and lock-freedom *together* for fine-grained concurrent objects, and verifying *full* correctness of optimizations of concurrent algorithms.

End-to-End Verification of Stack-Space Bounds for C Programs (PLDI'14) Verified compilers guarantee the preservation of semantic properties and thus enable formal verification of programs at the source level. However, important quantitative properties such as memory and time usage still have to be verified at the machine level where interactive proofs tend to be more tedious and automation is more challenging. In this work, the PI and his team develop a new framework that enables the formal verification of stack-space bounds of compiled machine code at the C level. It consists of a verified CompCert-based compiler that preserves quantitative properties, a verified quantitative program logic for interactive stack-bound development, and a verified stack analyzer that automatically derives stack bounds during compilation.

The framework is based on event traces that record function calls and returns. The source language is CompCert Clight and the target language is x86 assembly. The compiler is implemented in the Coq Proof Assistant and it is proved that crucial properties of event traces are preserved during compilation. A novel quantitative Hoare logic is developed to verify stack-space bounds at the CompCert Clight level. The quantitative logic is implemented in Coq and proved sound with respect to event traces generated by the small-step semantics of CompCert Clight. Stack-space bounds can be proved at the source level without taking into account low-level details that depend on the implementation of the compiler. The compiler fills in these low-level details during compilation and generates a concrete stack-space bound that applies to the produced machine code. The verified stack analyzer is guaranteed to automatically derive bounds for code with non-recursive functions. It generates a derivation in the quantitative logic to ensure soundness as well as interoperability with interactively developed stack bounds. In an experimental evaluation, the developed framework is used to obtain verified stack-space bounds for micro benchmarks as well as real system code. The examples include the verified operating-system kernel CertiKOS, parts of the MiBench embedded benchmark suite, and programs from the CompCert benchmarks. The derived bounds are close to the measured stack-space usage of executions of the compiled programs on a Linux x86 system.

A Separation Logic for Enforcing Declarative Information Flow Control Policies (POST'14) In this work, the PI and his student develop a new program logic for proving that a program does not release information about sensitive data in an unintended way. The most important feature of the logic is that it provides a formal security guarantee while supporting "declassification policies" that describe precise conditions under which a piece of sensitive data can be released. They leverage the power of Hoare Logic to express the policies and security guarantee in terms of state predicates. This allows their system to be far more specific regarding declassification conditions than most

other information flow systems. The logic is designed for reasoning about a C-like, imperative language with pointer manipulation and aliasing. They therefore make use of ideas from Separation Logic to reason about data in the heap.

Characterizing Progress Properties of Concurrent Objects via Contextual Refinements (CONCUR'13) Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, or deadlock-freedom. Conventional informal or semi-formal definitions of these progress properties describe conditions under which a method call is guaranteed to complete, but it is unclear how these demotions can be utilized to formally verify system software in a layered and modular way. In this work, the PI and his team propose a unified framework based on contextual refinements to show exactly how progress properties aect the behaviors of client programs. They give formal operational definitions of all common progress properties and prove that for linearizable objects, each progress property is equivalent to a specific type of contextual refinement that preserves termination. The equivalence ensures that verification of such a contextual refinement for a concurrent object guarantees both linearizability and the corresponding progress property. Contextual refinement also enables them to verify safety and liveness properties of client programs at a high abstraction level by soundly replacing concrete method implementations with abstract atomic operations.

Quantitative Reasoning for Proving Lock-Freedom (LICS'13) In this work, the PI and his team present a novel quantitative proof technique for the modular and local verification of lock-freedom. In contrast to proofs based on temporal rely-guarantee requirements, this new quantitative reasoning method can be directly integrated in modern program logics that are designed for the verification of safety properties. Using a single formalism for verifying memory safety and lock-freedom allows a combined correctness proof that verifies both properties simultaneously. This work presents one possible formalization of this quantitative proof technique by developing a variant of concurrent separation logic (CSL) for total correctness. To enable quantitative reasoning, CSL is extended with a predicate for affine tokens to account for, and provide an upper bound on the number of loop iterations in a program. Lock-freedom is then reduced to total-correctness proofs. Quantitative reasoning is demonstrated in detail, both informally and formally, by verifying the lockfreedom of Treiber's non-blocking stack. Furthermore, it is shown how the technique is used to verify the lock-freedom of more advanced shared-memory data structures that use elimination backoff schemes and hazard-pointers.

Compositional Verification of a Baby Virtual Memory Manager (CPP'12) A virtual memory manager (VMM) is a part of an operating system that provides the rest of the kernel with an abstract model of memory. Although small in size, it involves complicated and interdependent invariants that make monolithic verification of the VMM and the kernel running on top of it difficult. In this work, the PI and his team make the observation that a VMM is constructed in layers: physical page

allocation, page table drivers, address space API, etc., each layer providing an abstraction that the next layer utilizes. They use this layering to simplify the verification of individual modules of VMM and then to link them together by composing a series of small refinements. The compositional verification also supports function calls from less abstract layers into more abstract ones, allowing us to simplify the verification of initialization functions as well. To facilitate such compositional verification, they develop a framework that assists in creation of verification systems for each layer and refinements between the layers. Using this framework, they have produced a certification of BabyVMM, a small VMM designed for simplified hardware. The same proof also shows that a certified kernel using BabyVMM's virtual memory abstraction can be refined following a similar sequence of refinements, and can then be safely linked with BabyVMM. Both the verification framework and the entire certification of BabyVMM have been mechanized in the Coq Proof Assistant.

A Case for Behavior-Preserving Actions in Separation Logic (APLAS'12) Separation Logic is a widely-used tool that allows for local reasoning about imperative programs with pointers. A straightforward definition of this "local reasoning" is that, whenever a program runs safely on some state, any additional state has no effect on the program's behavior. In the presence of nondeterminism, however, local reasoning must be defined as something more subtle; specifically, additional state is allowed to decrease the amount of nondeterminism of the program. This subtlety causes difficulty in proving various metatheoretical facts about Separation Logic and its variants. Four specific examples are: (1) specifying the behavior of a program on its minimal footprint does not provide a complete specification; (2) data refinement requires a rather unintuitive restriction that the memory used by an abstract module be a subset of the memory used by a concrete module refining the abstract one; (3) Relational Separation Logic requires quite a bit of additional work to prove the frame rule sound; and (4) it is quite tricky to define a model of Separation Logic in which the total domain of memory is finite. In this work, the PI and his student show how to cleanly resolve all of these issues by strengthening the definition of local reasoning to eliminate the subtlety. They contend that this solution will also similarly resolve future metatheoretical issues.

Modular Verification of Concurrent Thread Management (APLAS'12) Thread management is an essential functionality in OS kernels. However, verification of thread management remains a challenge, due to two conflicting requirements: on the one hand, a thread manager—operating below the thread abstraction layer–should hide its implementation details and be verified independently from the threads being managed; on the other hand, the thread management code in many real-world systems is concurrent, which might be executed by the threads being managed, so it seems inappropriate to abstract threads away in the verification of thread managers. Previous approaches on kernel verification view thread managers as sequential code, thus cannot be applied to thread management in realistic kernels. In this work, the PI and his team propose a novel two-layer framework to verify concurrent thread management. They choose a lower abstraction level than

the previous approaches, where they abstract away the context switch routine only, and allow the rest of the thread management code to run concurrently in the upper level. They also treat thread management data as abstract resources so that threads in the environment can be specified in assertions and be reasoned about in a proof system similar to concurrent separation logic.

VeriML: A Dependently-Typed, User-Extensible, and Language-Centric Approach to Proof Assistant Software certification is a promising approach to producing programs which are virtually free of bugs. It requires the construction of a formal proof which establishes that the code in question will behave according to its specification — a higher-level description of its functionality. The construction of such formal proofs is carried out in tools called proof assistants. Advances in the current state-of-the-art proof assistants have enabled the certification of a number of complex and realistic systems software.

Despite such success stories, large-scale proof development is an arcane art that requires significant manual effort and is extremely time-consuming. The widely accepted best practice for limiting this effort is to develop domain-specific automation procedures to handle all but the most essential steps of proofs. Yet this practice is rarely followed or needs comparable development effort as well. This is due to a profound architectural shortcoming of existing proof assistants: developing automation procedures is currently overly complicated and error-prone. It involves the use of an amalgam of extension languages, each with a different programming model and a set of limitations, and with significant interfacing problems between them.

This thesis by Antonis Stampoulis (supervised by the PI) posits that this situation can be significantly improved by designing a proof assistant with extensibility as the central focus. Towards that effect, Stampoulis and the PI have designed a novel programming language called VeriML, which combines the benefits of the different extension languages used in current proof assistants while eschewing their limitations. The key insight of the VeriML design is to combine a rich programming model with a rich type system, which retains at the level of types information about the proofs manipulated inside automation procedures. The effort required for writing new automation procedures is significantly reduced by leveraging this typing information accordingly.

They show that generalizations of the traditional features of proof assistants are a direct consequence of the VeriML design. Therefore the language itself can be seen as the proof assistant in its entirety and also as the single language the user has to master. Also, they show how traditional automation mechanisms offered by current proof assistants can be programmed directly within the same language; users are thus free to extend them with domain-specific sophistication of arbitrary complexity. In the dissertation they present all aspects of the VeriML language: the formal definition of the language; an extensive study of its metatheoretic properties; the details of a complete prototype implementation; and a number of examples implemented and tested in the language.

Static and User-Extensible Proof Checking (POPL'12) Despite recent successes, large-scale proof development within proof assistants remains an arcane art that is extremely time-consuming. The PI and his team argue that this can be attributed to two profound shortcomings in the architecture of modern proof assistants. The first is that proofs need to include a large amount of minute detail; this is due to the rigidity of the proof checking process, which cannot be extended with domain-specific knowledge. In order to avoid these details, they rely on developing and using tactics, specialized procedures that produce proofs. Unfortunately, tactics are both hard to write and hard to use, revealing the second shortcoming of modern proof assistants. This is because there is no static knowledge about their expected use and behavior. As has recently been demonstrated, languages that allow type-safe manipulation of proofs, like Beluga, Delphin and VeriML, can be used to partly mitigate this second issue, by assigning rich types to tactics. Still, the architectural issues remain. In this work, the PI and his team build on this existing work, and demonstrate two novel ideas: an extensible conversion rule and support for static proof scripts. Together, these ideas enable us to support both user-extensible proof checking, and sophisticated static checking of tactics, leading to a new point in the design space of future proof assistants. Both ideas are based on the interplay between a light-weight staging construct and the rich type information available.

5 CONCLUSIONS

Operating System (OS) kernels form the bedrock of all system software—they can have the greatest impact on the resilience, extensibility, and security of today's computing hosts. A single kernel bug can easily wreck the entire system's integrity and protection. During the last four years, the PI and his team at Yale have successfully designed and implemented a clean-slate CertiKOS hypervisor kernel that runs on Intel and AMD multicore platforms and supports Linux and ROS applications on Landshark UGVs with good performance. They have also developed new certified programming methodologies and tools that support programming and composing certified abstraction layers (in C and assembly) and verify contextual safety, correctness, liveness, and security properties in one unified setting. They developed a fully specificied and verified single-core mCertiKOS kernel in Coq that is highly compositional with formally specified layers and strong contextual correctness guarantees. They also developed new semantics and logics for reasoning about declarative and decentralized information flow control with declassification, new certified resource anlaysis tools, and new logics for verifying safety and liveness properties of fine-grained concurrent programs. Finally, they developed new proof automation support including the design and implementation of the VeriML language and new Coq Ltac libraries.

Traditional OS kernels use a hardware-enforced "red line" to isolate the behaviors of user programs and to protect the integrity of the kernel code. The PI and his team's new layered approach to certified kernels replaces the red line with a large number of abstraction layers enforced via formal specification and proofs. They believe this will open up a whole new dimension of research efforts toward building truly reliable, secure, and extensible system software.

6 REFERENCES

[1] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. Lessons learned from microkernel verification—specification is the new bottleneck. In *Proceedings of Systems Software Verification Conference (SSV 2012)*, pages 18–32, 2012.

[2] B. N. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating System Principles*, 1995.

[3] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI '14: 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[4] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *PLDI '15: 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[5] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. 2011 ACM Conference on Programming Language Design and Implementation*, 2011.

[6] D. Costanzo and Z. Shao. A separation logic for enforcing declarative information flow control policies. In *Proc. 3rd Conference on Principles of Security and Trust (POST'14)*, volume 8414 of *LNCS*, pages 179–198. Springer-Verlag, Apr. 2014.

[7] M. Daum, N. Billing, and G. Klein. Concerned with the unprivileged: user programs in kernel extensions. *Formal Aspects of Computing*, 26(6):1205–1229, 2014.

[8] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 206–214, Jan. 1977.

[9] K. Elphinstone and G. Heiser. From l3 to sel4, what have we learnt in 20 years of l4 microkernels? In *Proc. 2013 ACM Symposium on Operating System Principles (SOSP)*, pages 133–150, 2013.

[10] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, Dec. 1995.

[11] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *VSTTE'08*, pages 54–69, 2008.

[12] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reasoning*, 42(2-4):301–347, 2009.

[13] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[14] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 595–608, 2015.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Apr. 2008.

[16] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[17] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[18] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. McCarthy, J. Rafkind, S. Tobin-stadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proc. 39th ACM Symposium on Principles of Programming Languages*, 2012.

[19] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), Feb. 2014.

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[21] X. Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2013.

[22] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[23] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The compcert memory model, version 2. Technical Report RR-7987, INRIA, 2012.

[24] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformation. *Journal of Automated Reasoning*, 2008.

[25] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *Proc. 29th IEEE Symposium on Logic in Computer Science*, page no. 65, July 2014.

[26] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *Proc. 24th International Conference on Concurrency Theory (CONCUR'13)*. Springer-Verlag, 2013.

[27] J. Liedtke. On micro-kernel construction. In *15th ACM Symposium on Operating System Principles*, 1995.

[28] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[29] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[30] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *Proc. IEEE Symposium on Security and Privacy*, pages 415–429, 2013.

[31] Z. Ni, D. Yu, and Z. Shao. Using xcap to certify realistic system code: Machine context management. In *Proc. 20th International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 189–206. Springer-Verlag, Sept. 2007.

[32] W. Paul, M. Broy, and T. In der Rieden. The verisoft project. URL: `http://www.verisoft.de`, 2006.

[33] T. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *PLDI13*, pages 471–482, 2013.

[34] T. Sewell, S. Winwood, P. Gammie, T. C. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *ITP*, pages 325–340, 2011.

[35] Z. Shao. Certified software. *Communications of the ACM*, 53(12):56–66, December 2010.

[36] Z. Shao and B. Ford. Advanced development of certified os kernels. Technical Report YALEU/DCS/TR-1436, Dept. of Computer Science, Yale University, New Haven, CT, July 2010.

[37] A. Stampoulis. *VeriML: A Dependently-Typed, User-Extensible, and Language-Centric Approach to Proof Assistant*. PhD thesis, Department of Computer Science, Yale University, November 2012.

[38] A. Stampoulis and Z. Shao. VeriML:typed computation of logical terms inside a language with effects. In *Proc. 2010 ACM SIGPLAN International Conference on Functional Programming*, pages 333–344, 2010. `flint.cs.yale.edu/publications/veriml.html`.

[39] A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, New York, NY, USA, 2012. ACM.

[40] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2012.

[41] A. Vaynberg and Z. Shao. Compositional verification of a baby virtual memory manager. In *Proc. 2nd International Conf. on Certified Programs and Proofs*, pages 143–159, Dec 2012.

[42] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, pages 263–278, 2006.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
APLAS	Asian Symposium on Programming Languages and Systems
CONCUR	Conference on Concurrency Theory
CPP	Certified Programs and Proofs
CRASH	Clean-Slate Design of Resilient, Adaptive, Secure Hosts
DIFC	Declarative Decentralized Information Flow Control
EIP	Extended Instruction Pointer register
ESOP	European Symposium on Programming
ESP	Extended Stack Pointer register
EPT	Extended Page Table
HACMS	High-Assurance Cyber Military Systems
IPC	Inter-Process Communication
KVM	Kernel-based Virtual Machine
LICS	Logic In Computer Science
MMU	memory management unit
OS	Operating System
PI	Primary Investigator
PLDI	Programming Language Design and Implementation
pm	Person Months
POPL	Symposium on Principles of Programming Languages
POST	Principles of Security and Trust
ROS	Robot Operating System
VMCS	Virtual Machine Control Structure
VMM	Virtual Memory Manager
VMX	Virtual Machine eXtension

APPENDIX

Here are a list of important and representative publications produced by the PI and his team during the funding period of this DARPA CRASH project.

1. **POPL12:** Static and User-Extensible Proof Checking (pages 40-159)
2. **APLAS12a:** A Case for Behavior-Preserving Actions in Separation Logic. (pages 160-177)
3. **APLAS12b:** Modular Verification of Concurrent Thread Management. (pages 178-194)
4. **CPP12:** Compositional Verification of a Baby Virtual Memory Manager. (pages 195-210)
5. **LICS13:** Quantitative Reasoning for Proving Lock-Freedom. (pages 211-228)
6. **CONCUR13:** Characterizing Progress Properties of Concurrent Objects via Contextual Refinements. (pages 229-309)
7. **POST14:** A Separation Logic for Enforcing Declarative Information Flow Control Policies. (pages 310-330)
8. **PLDI14:** End-to-End Verification of Stack-Space Bounds for C Programs. (pages 331-342)
9. **LICS14:** Compositional Verification of Termination-Preserving Refinement of Concurrent Programs. (pages 343-434)
10. **CPP15:** A Compositional Semantics for Verified Separate Compilation and Linking. (pages 435-446)
11. **POPL15:** Deep Specifications and Certified Abstraction Layers. (pages 447-460)
12. **ESOP15:** Automatic Static Cost Analysis for Parallel Programs. (pages 461-486)
13. **PLDI15:** Compositional Certified Resource Bounds. (pages 487-498)

Static and User-Extensible Proof Checking

Antonis Stampoulis Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520, USA
{antonis.stampoulis,zhong.shao}@yale.edu

Abstract

Despite recent successes, large-scale proof development within proof assistants remains an arcane art that is extremely time-consuming. We argue that this can be attributed to two profound shortcomings in the architecture of modern proof assistants. The first is that proofs need to include a large amount of minute detail; this is due to the rigidity of the proof checking process, which cannot be extended with domain-specific knowledge. In order to avoid these details, we rely on developing and using tactics, specialized procedures that produce proofs. Unfortunately, tactics are both hard to write and hard to use, revealing the second shortcoming of modern proof assistants. This is because there is no static knowledge about their expected use and behavior.

As has recently been demonstrated, languages that allow type-safe manipulation of proofs, like Beluga, Delphin and VeriML, can be used to partly mitigate this second issue, by assigning rich types to tactics. Still, the architectural issues remain. In this paper, we build on this existing work, and demonstrate two novel ideas: an *extensible conversion rule* and support for *static proof scripts*. Together, these ideas enable us to support both user-extensible proof checking, and sophisticated static checking of tactics, leading to a new point in the design space of future proof assistants. Both ideas are based on the interplay between a light-weight staging construct and the rich type information available.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Verification

1. Introduction

There have been various recent successes in using proof assistants to construct foundational proofs of large software, like a C compiler [Leroy 2009] and an OS microkernel [Klein et al. 2009], as well as complicated mathematical proofs [Gonthier 2008]. Despite this success, the process of large-scale proof development using the foundational approach remains a complicated endeavor that requires significant manual effort and is plagued by various architectural issues.

The big benefit of using a foundational proof assistant is that the proofs involved can be checked for validity using a very small proof checking procedure. The downside is that these proofs are very large, since proof checking is fixed. There is no way to add domain-specific knowledge to the proof checker, which would enable proofs that spell out less details. There is good reason for this, too: if we allowed arbitrary extensions of the proof checker, we could very easily permit it to accept invalid proofs.

Because of this lack of extensibility in the proof checker, users rely on tactics: procedures that produce proofs. Users are free to write their own tactics, that can create domain-specific proofs. In fact, developing domain-specific tactics is considered to be good engineering when doing large developments, leading to significantly decreased overall effort – as shown, e.g. in Chlipala [2011]. Still, using and developing tactics is error-prone. Tactics are essentially untyped functions that manipulate logical terms, and thus tactic programming is untyped. This means that common errors, like passing the wrong argument, or expecting the wrong result, are not caught statically. Exacerbating this, proofs contained within tactics are not checked statically, when the tactic is defined. Therefore, even if the tactic is used correctly, it could contain serious bugs that manifest only under some conditions.

With the recent advent of programming languages that support strongly typed manipulation of logical terms, such as Beluga [Pientka and Dunfield 2008], Delphin [Poswolsky and Schürmann 2008] and VeriML [Stampoulis and Shao 2010], this situation can be somewhat mitigated. It has been shown in Stampoulis and Shao [2010] that we can specify what kinds of arguments a tactic expects and what kind of proof it produces, leading to a type-safe programming style. Still, this does not address the fundamental problem of proof checking being fixed – users still have to rely on using tactics. Furthermore, the proofs contained within the type-safe tactics are in fact proof-producing programs, which need to be evaluated upon invocation of the tactic. Therefore proofs within tactics are not checked statically, and they can still cause the tactics to fail upon invocation.

In this paper, we build on the past work on these languages, aiming to solve both of these issues regarding the architecture of modern proof assistants. We introduce two novel ideas: support for an **extensible conversion rule** and **static proof scripts** inside tactics. The former technique enables proof checking to become user-extensible, while maintaining the guarantee that only logically sound proofs are admitted. The latter technique allows for statically checking the proofs contained within tactics, leading to increased guarantees about their runtime behavior. Both techniques are based on the same mechanism, which consists of a light-weight staging construct. There is also a deep synergy between them, allowing us to use the one to the benefit of the other.

Our main contributions are the following:

- First, we present what we believe is the first technique for having an extensible conversion rule, which combines the following characteristics: it is safe, meaning that it preserves logical soundness; it is user-extensible, using a familiar, generic pro-

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

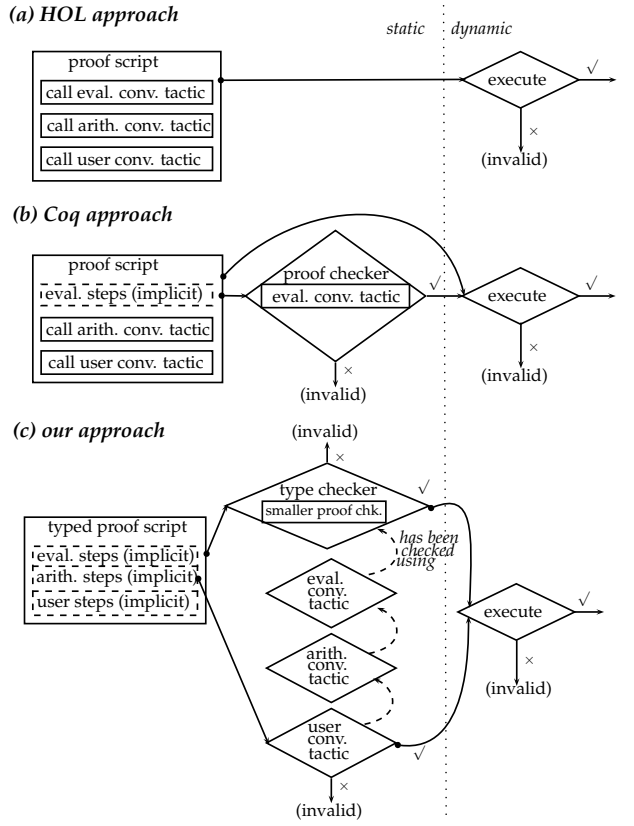


Figure 1. Checking proof scripts in various proof assistants

gramming model; and, it does not require metatheoretic additions to the logic, but can be used to simplify the logic instead.

- Second, building on existing work for typed tactic development, we introduce static checking of the proof scripts contained within tactics. This significantly reduces the development effort required, allowing us to write tactics that benefit from existing tactics and from the rich type information available.
- Third, we show how typed proof scripts can be seen as an alternative form of proof witness, which falls between a proof object and a proof script. Receivers of the certificate are able to decide on the tradeoff between the level of trust they show and the amount of resources needed to check its validity.

In terms of technical contributions, we present a number of technical advances in the metatheory of the aforementioned programming languages. These include a simple staging construct that is crucial to our development and a new technique for variable representation. We also show a condition under which static checking of proof scripts inside tactics is possible. Last, we have extended an existing prototype implementation with a significant number of features, enabling it to support our claims, while also rendering its use as a proof assistant more practical.

2. Informal presentation

Glossary of terms. We will start off by introducing some concepts that will be used throughout the paper. The first fundamental concept we will consider is the notion of a *proof object*: given a derivation of a proposition inside a formal logic, a proof object is a term representation of this derivation. A *proof checker* is a program that can decide whether a given proof object is a valid derivation of a specific proposition or not. Proof objects are extremely verbose and are thus hard to write by hand. For this reason, we use *tactics*: functions that produce proof objects. By combining tactics together, we create proof-producing programs, which we call *proof scripts*. If a proof script is evaluated, and the evaluation completes successfully, the resulting proof object can be checked using the original proof checker. In this way, the trusted base of the system is kept at the absolute minimum. The language environment where proof scripts and tactics are written and evaluated is called a *proof assistant*; evidently, it needs to include a proof checker.

Checking proof objects. In order to keep the size of proof objects manageable, many of the logics used for mechanized proof checking include a *conversion rule*. This rule is used implicitly by the proof checker to decide whether any two propositions are equivalent; if it determines that they are indeed so, the proof of their equivalence can be omitted. We can thus think of it as a special tactic that is embedded within the proof checker, and used implicitly.

The more sophisticated the relation supported by the conversion rule is, the simpler are proof objects to write, since more details can be omitted. On the other hand, the proof checker becomes more complicated, as does the metatheory proof showing the soundness of the associated logic. The choice in Coq [Barras et al. 2010], one of the most widely used proof assistants, with respect to this trade-off, is to have a conversion rule that identifies propositions up to evaluation. Nevertheless, extended notions of conversion are desirable, leading to proposals like CoqMT [Strub 2010], where equivalence up to first-order theories is supported. In both cases, the conversion rule is fixed, and extending it requires significant amounts of work. It is thus not possible for users to extend it using their own, domain-specific tactics, and proof objects are thus bound to get large. This is why we have to resort to writing proof scripts.

Checking proof scripts. As mentioned earlier, in order to validate a proof script we need to evaluate it (see Fig. 1a); this is the modus operandi in proof assistants of the HOL family [Harrison 1996; Slind and Norrish 2008]. Therefore, it is easy to extend the checking procedure for proof scripts by writing a new tactic, and calling it as part of a script. The price that this comes to is that there is no way to have any sort of static guarantee about the validity of the script, as proof scripts are completely untyped. This can be somewhat mitigated in Coq by utilizing the static checking that it already supports: the proof checker, and especially, the conversion rule it contains (see Fig. 1b). We can employ proof objects in our scripts; this is especially useful when the proof objects are trivial to write but trigger complex conversion checks. This is the essential idea behind techniques like proof-by-reflection [Boutin 1997], which lead to more robust proof scripts.

In previous work [Stampoulis and Shao 2010] we introduced VeriML, a language that enables programming tactics and proof scripts in a typeful manner using a general-purpose, side-effectful programming model. Combining typed tactics leads to *typed proof scripts*. These are still programs producing proof objects, but the proposition they prove is carried within their type. Information about the current proof state (the set of hypotheses and goals) is also available statically at every intermediate point of the proof script. In this way, the static assurances about proof scripts are significantly increased and many potential sources of type errors are removed. On the other hand, the proof objects contained within the scripts are still checked using a fixed proof checker; this ultimately means that the set of possible static guarantees is still fixed.

Extensible conversion rule. In this paper, we build on our earlier work on VeriML. In order to further increase the amount of static checking of proof scripts that is possible within this language, we propose the notion of an extensible conversion rule (see Fig. 1c). It enables users to write their own domain-specific conversion checks

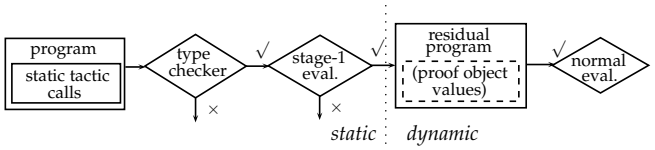


Figure 2. Staging in VeriML

$t ::=$ *proof object constructors* $\mid$ *propositions*
$\quad \mid$ *natural numbers, lists, etc.* $\mid$ *sorts and types* $\mid X/\sigma$
$\Phi ::= \bullet \mid \Phi, x : t \qquad T ::= [\Phi]t$
$\Psi ::= \bullet \mid \Psi, X : T \qquad \sigma ::= \bullet \mid \sigma, x \mapsto t$
main judgement: $\Psi; \Phi \vdash t : t'$ *(type of a logical term)*

Figure 3. Assumptions about the logic language

that get included in the conversion rule. This leads to simpler proof scripts, as more parts of the proof can be inferred by the conversion rule and can therefore be omitted. Also, it leads to increased static guarantees for proof scripts, since the conversion checks happen before the rest of the proof script is evaluated.

The way we achieve this is by programming the conversion checks as type-safe tactics within VeriML, and then evaluating them statically using a simple staging mechanism (see Fig. 2). The type of the conversion tactics requires that they produce a proof object which proves the claimed equivalence of the propositions. In this way, type safety of VeriML guarantees that soundness is maintained. At the same time, users are free to extend the conversion rule with their own conversion tactics written in a familiar programming model, without requiring any metatheoretic additions or termination proofs. Such proofs are only necessary if decidability of the extra conversion checks is desired. Furthermore, this approach allows for metatheoretic reductions as the original conversion rule can be programmed within the language. Thus it can be removed from the logic, and replaced by the simpler notion of explicit equalities, leading to both simpler metatheory and a smaller trusted base.

Checking tactics. The above approach addresses the issue of being able to extend the amount of static checking possible for proof scripts. But what about tactics? Our existing work on VeriML shows how the increased type information addresses some of the issues of tactic development using current proof assistants, where tactics are programmed in a completely untyped manner.

Still, if we consider the case of tactics more closely, we will see that there is a limitation to the amount of checking that is done statically, even using this language. When programming a new tactic, we would like to reuse existing tactics to produce the required proofs. Therefore, rather than writing proof objects by hand inside the code of a tactic, we would rather use proof scripts. The issue is that in order to check whether the contained proof scripts are valid, they need to be evaluated – but this only happens when an invocation of the tactic reaches the point where the proof script is used. Therefore, the static guarantees that this approach provides are severely limited by the fact that the proof scripts inside the tactics cannot be checked statically, when the tactic is defined.

Static proof scripts. This is the second fundamental issue we address in this paper. We show that the same staging construct utilized for introducing the extensible conversion rule, can be leveraged to perform *static proof checking for tactics*. The crucial point of our approach is the proof of existence of a transformation between proof objects, which suggests that under reasonable conditions, a proof script contained within a tactic can be transformed into a static proof script. This static script can then be evaluated at tactic definition time, to be checked for validity.

Last, we will show that this approach lends itself well to writing extensions of the conversion rule. We show that we can create a layering of conversion rules: using a basic conversion rule as a starting point, we can utilize it inside static proof scripts to implicitly prove the required obligations of a more advanced version, and so on. This minimizes the required user effort for writing new conversion rules, and enables truly modular proof checking.

3. Our toolbox

In this section, we will present the essential ingredients that are needed for the rest of our development. The main requirement is a language that supports type-safe manipulation of terms of a particular logic, as well as a general-purpose programming model that includes general recursion and other side-effectful operations. Two recently proposed languages for manipulating LF terms, Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008], fit this requirement, as does VeriML [Stampoulis and Shao 2010], which is a language used to write type-safe tactics. Our discussion is focused on the latter, as it supports a richer ML-style calculus compared to the others, something useful for our purposes. Still, our results apply to all three.

We will now briefly describe the constructs that these languages support, as well as some new extensions that we propose. The interested reader can read more about these constructs in Sec. 6 and in our technical report [Stampoulis and Shao 2012].

A formal logic. The computational language we are presenting is centered around manipulation of terms of a specific formal logic. We will see more details about this logic in Sec. 4. For the time being, it will suffice to present a set of assumptions about the syntactic classes and typing judgements of this logic, shown in Fig. 3. Logical terms are represented by the syntactic class t, and include proof objects, propositions, terms corresponding to the domain of discourse (e.g. natural numbers), and the needed sorts and type constructors to classify such terms. Their variables are assigned types through an ordered context Φ. A package of a logical term t together with the variables context it inhabits Φ is called a contextual term and denoted as $T = [\Phi]t$. Our computational language works over contextual terms for reasons that will be evident later. The logic incorporates such terms by allowing them to get substituted for *meta-variables* X, using the constructor X/σ. When a term $T = [\Phi']t$ gets substituted for X, we go from the Φ' context to the current context Φ using the substitution σ.

Logical terms are classified using other logical terms, based on the normal variables environment Φ, and also an environment Ψ that types meta-variables, thus leading to the $\Psi; \Phi \vdash t : t'$ judgement. For example, a term t representing a closed proposition will be typed as $\bullet; \bullet \vdash t : \mathsf{Prop}$, while a proof object t_{pf} proving that proposition will satisfy the judgement $\bullet; \bullet \vdash t_{\text{pf}} : t$.

ML-style functional programming. We move on to the computational language. As its main core, we assume an ML-style functional language, supporting general recursion, algebraic data types and mutable references (see Fig. 4). Terms of this fragment are typed under a computational variables environment Γ and a store typing environment Σ, mapping mutable locations to types. Typing judgements are entirely standard, leading to a $\Sigma; \Gamma \vdash e : \tau$ judgement for typing expressions.

Dependently-typed programming over logical terms. As shown in Fig. 5, the first important additions to the ML computational core are constructs for dependent functions and products over contextual terms T. Abstraction over contextual terms is denoted as $\lambda X : T.e$. It has the dependent function type $(X : T) \to \tau$. The type is dependent since the introduced logical term might be used as the type of

$$
\begin{array}{l}
k ::= * \mid k_1 \to k_2 \\
\tau ::= \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha : k.\tau \\
\quad \mid \forall\alpha : k.\tau \mid \alpha \mid \mathsf{array}\ \tau \mid \lambda\alpha : k.\tau \mid \tau_1\ \tau_2 \mid \cdots \\
e ::= () \mid n \mid e_1 + e_2 \mid e_1 \le e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else} e_2 \\
\quad \mid \lambda\mathbf{x} : \tau.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e \\
\quad \mid \mathsf{case}(e, \mathbf{x}_1.e_1, \mathbf{x}_2.e_2) \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \Lambda\alpha : k.e \mid e\ \tau \\
\quad \mid \mathsf{fix}\ \mathbf{x} : \tau.e \mid \mathsf{mkarray}(e, e') \mid e[e'] \mid e[e'] := e'' \mid l \mid \mathsf{error} \mid \cdots \\
\Gamma ::= \bullet \mid \Gamma, \mathbf{x} : \tau \mid \Gamma, \alpha : k \qquad \Sigma ::= \bullet \mid \Sigma, l : \mathsf{array}\ \tau
\end{array}
$$

Figure 4. Syntax for the computational language (ML fragment)

$$
\begin{array}{l}
\tau ::= \cdots \mid (X : T) \to \tau \mid (X : T) \times \tau \mid (\phi : \mathsf{ctx}) \to \tau \\
e ::= \cdots \mid \lambda X : T.e \mid e\ T \mid \lambda\phi : \mathsf{ctx}.e \mid e\ \Phi \mid \langle T, e \rangle \\
\quad \mid \mathsf{let}\ \langle X, \mathbf{x} \rangle = e\ \mathsf{in}\ e' \\
\quad \mid \mathsf{holcase}\ T\ \mathsf{return}\ \tau\ \mathsf{of}\ (T_1 \mapsto e_1) \cdots (T_n \mapsto e_n) \\
\quad \mid \mathsf{ctxcase}\ \Phi\ \mathsf{return}\ \tau\ \mathsf{of}\ (\Phi_1 \mapsto e_1) \cdots (\Phi_n \mapsto e_n)
\end{array}
$$

Figure 5. Syntax for the computational language (logical term constructs)

another term. An example would be a function that receives a proposition plus a proof object for that proposition, with type: $(P : \mathsf{Prop}) \to (X : P) \to \tau$. Dependent products that package a contextual logical term with an expression are introduced through the $\langle T, e \rangle$ construct and eliminated using $\mathsf{let}\ \langle X, \mathbf{x} \rangle = e\ \mathsf{in}\ e'$; their type is denoted as $(X : T) \times \tau$. Especially for packages of proof objects with the unit type, we introduce the syntax $\mathsf{LT}(T)$.

Last, in order to be able to support functions that work over terms in any context, we introduce context polymorphism, through a similarly dependent function type over contexts. With these in mind, we can define a simple tactic that gets a packaged proof of a universally quantified formula, and an instantiation term, and returns a proof of the instantiated formula as follows:

$$
\begin{array}{l}
\mathsf{instantiate} : \ (\phi : \mathsf{ctx},\ T : [\phi]\,\mathsf{Type},\ P : [\phi,\ x : T]\,\mathsf{Prop},\ a : [\phi]\,T) \to \\
\qquad \mathsf{LT}([\phi]\,\forall x : T, P) \to \mathsf{LT}([\phi]\,P/[\mathsf{id}_\phi,\ a]) \\
\mathsf{instantiate}\ \phi\ T\ P\ a\ \mathsf{pf}\ = \mathsf{let}\ \langle H \rangle\ = \mathsf{pf}\ \mathsf{in}\ \langle H\ a \rangle
\end{array}
$$

From here on, we will omit details about contexts and substitutions in the interest of presentation.

Pattern matching over terms. The most important new construct that VeriML supports is a pattern matching construct over logical terms denoted as holcase. This construct is used for dependent matching of a logical term against a set of patterns. The return clause specifies its return type; we omit it when it is easy to infer. Patterns are normal terms that include unification variables, which can be present under binders. This is the essential reason why contextual terms are needed.

Pattern matching over environments. For the purposes of our development, it is very useful to support one more pattern matching construct: matching over logical variable contexts. When trying to construct a certain proof, the logical environment represents what the current proof context is: what the current logical hypotheses at hand are, what types of terms have been quantified over, etc. By being able to pattern match over the environment, we can "look up" things in our current set of hypotheses, in order to prove further propositions. We can thus view the current environment as representing a simple form of the current *proof state*; the pattern matching construct enables us to manipulate it in a type-safe manner.

One example is an "assumption" tactic, that tries to prove a proposition by searching for a matching hypotheses in the context:

$$
\begin{array}{l}
\mathsf{assumption} : \ (\phi : \mathsf{ctx}, P : \mathsf{Prop}) \to \mathsf{option}\ \mathsf{LT}(P) \\
\mathsf{assumption}\ \phi\ P = \\
\quad \mathsf{ctxcase}\ \phi\ \mathsf{of} \\
\qquad \phi',\ H : P \mapsto \mathsf{return}\ \langle H \rangle \\
\qquad \mid \phi',\ \_ \qquad \mapsto \mathsf{assumption}\ \phi'\ P
\end{array}
$$

Proof object erasure semantics (new feature). The only construct that can influence the evaluation of a program based on the structure of a logical term is the pattern matching construct. For our purposes, pattern matching on proof objects is not necessary – we never look into the structure of a completed proof. Thus we can have the typing rules of the pattern matching construct specifically disallow matching on proof objects.

In that case, we can define an alternate operational semantics for our language where all proof objects are *erased* before using the original small-step reduction rules. Because of type safety, these proof-erasure semantics are guaranteed to yield equivalent results: even if no proof objects are generated, they are still bound to exist.

Implicit arguments. Let us consider again the *instantiate* function defined earlier. This function expects five arguments. From its type alone, it is evident that only the last two arguments are strictly necessary. The last argument, corresponding to a proof expression for the proposition $\forall x : T, P$, can be used to reconstruct exactly the arguments ϕ, T and P. Furthermore, if we know what the resulting type of a call to the function needs to be, we can choose even the instantiation argument a appropriately. We employ a simple inferrence mechanism so that such arguments are omitted from our programs. This feature is also crucial in our development in order to implicitly maintain and utilize the current proof state within our proof scripts.

Minimal staging support (new feature). Using the language we have seen so far we are able to write powerful tactics using a general-purpose programming model. But what if, inside our programs, we have calls to tactics where all of their arguments are constant? Presumably, those tactic calls could be evaluated to proof objects prior to tactic invocation. We could think of this as a form of generalized constant folding, which has one intriguing benefit: we can tell statically whether the tactic calls succeed or not.

This paper is exactly about exploring this possibility. Towards this effect, we introduce a rudimentary staging construct in our computational language. This takes the form of a letstatic construct, which binds a static expression to a variable. The static expression is evaluated during stage one (see Fig. 2), and can only depend on other static expressions. Details of this construct are presented in Fig. 11d and also in Sec. 6. After this addition, expressions in our language have a three-phase lifetime, that are also shown in Fig. 2.

- type-checking, where the well-formedness of expressions according to the rules of the language is checked, and inference of implicit arguments is performed
- static evaluation, where expressions inside letstatic are reduced to values, yielding a residual expression
- run-time, where the residual expression is evaluated

4. Extensible conversion rule

With these tools at hand, let us now return to the first issue that motivates us: the fact that proof checking is rigid and cannot be extended with user-defined procedures. As we have said in our introduction, many modern proof assistants are based on logics that include a *conversion rule*. This rule essentially identifies proposi-

$$
\begin{array}{rrcl}
\text{(sorts)} & s & ::= & \mathsf{Type} \mid \mathsf{Type}' \\
\text{(kinds)} & \mathcal{K} & ::= & \mathsf{Prop} \mid \mathsf{Nat} \mid \mathcal{K}_1 \to \mathcal{K}_2 \\
\text{(props.)} & P & ::= & P_1 \to P_2 \mid \forall x : \mathcal{K}.P \mid x \mid \mathsf{True} \\
 & & & \mid \mathsf{False} \mid P_1 \wedge P_2 \mid \cdots \\
\text{(dom.obj.)} & d & ::= & \mathsf{Zero} \mid \mathsf{Succ}\ d \mid P \mid \cdots \\
\text{(proof objects)} & \pi & ::= & x \mid \lambda x : P.\pi \mid \pi_1\ \pi_2 \mid \lambda x : \mathcal{K}.\pi \\
 & & & \mid \pi\ d \mid \cdots \\
\text{(HOL terms)} & t & ::= & s \mid \mathcal{K} \mid P \mid d \mid \pi
\end{array}
$$

Selected rules:

$$
\frac{\Psi;\ \Phi, x : P \vdash \pi : P'}{\Psi;\ \Phi \vdash \lambda x : P.\pi : P \to P'}\ {\scriptstyle \to \textsc{Intro}}
\qquad
\frac{\Psi;\ \Phi \vdash \pi : P \to P' \qquad \Psi;\ \Phi \vdash \pi' : P}{\Psi;\ \Phi \vdash \pi\ \pi' : P'}\ {\scriptstyle \to \textsc{Elim}}
$$

Figure 6. Syntax and selected rules of the logic language λHOL

$$
\textsc{Conversion}\quad \frac{\Psi;\ \Phi \vdash_c \pi : P \qquad P =_{\beta\mathbb{N}} P'}{\Psi;\ \Phi \vdash_c \pi : P'}
$$

$d \to_{\beta\mathbb{N}} d'$

$$
\begin{array}{l}
(\lambda x : \mathcal{K}.d)\ d' \to_{\beta\mathbb{N}} d[d'/x] \\
\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ \mathsf{zero} \to_{\beta\mathbb{N}} d_z \\
\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ (\mathsf{succ}\ d) \to_{\beta\mathbb{N}} d_s\ d\ (\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ d)
\end{array}
$$

$d =_{\beta\mathbb{N}} d'$ is the compatible, reflexive, symmetric and transitive closure of $d \to_{\beta\mathbb{N}} d'$

Figure 7. Extending λHOL with the conversion rule ($\lambda\mathrm{HOL}_c$)

tions up to some equivalence relation: usually this is equivalence up to partial evaluation of the functions contained within propositions.

The supported relation is decided when the logic is designed. Any extension to this relation requires a significant amount of work, both in terms of implementation, and in terms of metatheoretic proof required. This is evidenced by projects that extend the conversion rule in Coq, such as Blanqui et al. [1999] and Strub [2010]. Even if user extensions are supported, those only take the form of first-order theories. Can we do better than this, enabling arbitrarily complex user extensions, written with the full power of ML, yet maintaining soundness?

It turns out that we can: this is the subject of this section. The key idea is to recognize that the conversion rule is essentially a tactic, embedded within the type checker of the logic. Calls to this tactic are made implicitly as part of checking a given proof object for validity. So how can we support a flexible, extensible alternative? Instead of hardcoding a conversion tactic within the logic type checker, we can program a type-safe version of the same tactic within VeriML, with the requirement that it provides proof of the claimed equivalence. Instead of calling the conversion tactic as part of proof checking, we use staging to call the tactic statically – after (VeriML) type checking, but before runtime execution. This can be viewed as a second, potentially non-terminating proof checking stage. Users are now free to write their own conversion tactics, extending the static checking available for proof objects and proof scripts. Still, soundness is maintained, since full proof objects in the original logic can always be constructed. As an example, we have extended the conversion rule that we use by a congruence closure procedure, which makes use of mutable data structures, and by an arithmetic simplification procedure.

4.1 Introducing: the conversion rule

First, let us present what the conversion rule really is in more detail. We will base our discussion on a simple type-theoretic higher-order

$$
\frac{\Psi;\ \Phi \vdash_e d_1 : \mathcal{K} \qquad \Psi;\ \Phi \vdash_c d_2 : \mathcal{K}}{\Psi;\ \Phi \vdash_e d_1 = d_2 : \mathsf{Prop}}
\qquad
\frac{\Psi;\ \Phi \vdash_e d : \mathcal{K}}{\Psi;\ \Phi \vdash_e \mathsf{refl}\ d : d = d}
$$

$$
\frac{\begin{array}{c}\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e P : \mathsf{Prop} \qquad \Psi;\ \Phi \vdash_e d_1 : \mathcal{K} \\ \Psi;\ \Phi \vdash_e \pi : P[d_1/x] \qquad \Psi;\ \Phi \vdash_e \pi' : d_1 = d_2\end{array}}{\Psi;\ \Phi \vdash_e \mathsf{leibniz}\ (\lambda x : \mathcal{K}.P)\ \pi\ \pi' : P[d_2/x]}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e \pi : d_1 = d_2}{\Psi;\ \Phi \vdash_e \mathsf{lamEq}\ (\lambda x : \mathcal{K}.\pi) : (\lambda x : \mathcal{K}.d_1) = (\lambda x : \mathcal{K}.d_2)}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e \pi : d_1 = d_2 \qquad \Psi;\ \Phi \vdash_e d_1 : \mathsf{Prop}}{\Psi;\ \Phi \vdash_e \mathsf{forallEq}\ (\lambda x : \mathcal{K}.\pi) : (\forall x : \mathcal{K}.d_1) = (\forall x : \mathcal{K}.d_2)}
$$

$$
\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e d : \mathcal{K}' \qquad \Psi;\ \Phi \vdash_e d' : \mathcal{K}}{\Psi;\ \Phi \vdash_e \mathsf{betaEq}\ (\lambda x : \mathcal{K}.d)\ d' : (\lambda x : \mathcal{K}.d)\ d' = d[d'/x]}
$$

Axioms assumed:

$$
\begin{array}{lcl}
\mathsf{natElimBase}_{\mathcal{K}} & : & \forall f_z.\forall f_s.\mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ \mathsf{zero} = f_z \\
\mathsf{natElimStep}_{\mathcal{K}} & : & \forall f_z.\forall f_s.\forall n.\ \mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ (\mathsf{succ}\ n) = \\
 & & \qquad f_s\ n\ (\mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ n)
\end{array}
$$

Figure 8. Extending λHOL with explicit equality ($\lambda\mathrm{HOL}_e$)

logic, based on the λHOL logic as described in Barendregt and Geuvers [1999], and used in our original work on VeriML [Stampoulis and Shao 2010]. We can think of such a logic composed by the following broad classes: the objects of the domain of discourse d, which are the objects that the logic reasons about, such as natural numbers and lists; their classifiers, the kinds $\mathcal{K}$ (classified in turn by sorts s); the propositions P; and the derivations, which prove that a certain proposition is true. We can represent derivations in a linear form as terms π in a typed lambda-calculus; we call such terms proof objects, and their types represent propositions in the logic. Checking whether a derivation is a valid proof of a certain proposition amounts to type-checking its corresponding proof object. Some details of this logic are presented in Fig. 6; the interested reader can find more information about it in the above references and in our technical report [Stampoulis and Shao 2012].

In Fig. 6, we show what the conversion rule looks like for this logic: it is a typing judgement that effectively identifies propositions up to an equivalence relation, with respect to checking proof objects. We call this version of the logic $\lambda\mathrm{HOL}_c$ and use $\vdash_c$ to denote its entailment relation. The equivalence relation we consider in the conversion rule is evaluation up to β-reductions and uses of primitive recursion of natural numbers, denoted as natElim. In this way, trivial arguments based on this notion of computation alone need not be witnessed, as for example is the fact that $(\mathsf{Succ}\ x) + y = \mathsf{Succ}\ (x + y)$ – when the addition function is defined by primitive recursion on the first argument. Of course, this is only a very basic use of the conversion rule. It is possible to omit larger proofs through much more sophisticated uses. This leads to simpler proofs and smaller proof objects.

Still, when using this approach, the choice of what relation is supported by the conversion rule needs to be made during the definition of the logic. This choice permeates all aspects of the metatheory of the logic. It is easy to see why, even with the tiny fragment of logic we have introduced. Most typing rules for proof objects in the logic are similar to the rules →Intro and →Elim: they are syntax-directed. This means that upon seeing the associated proof object constructor, like $\lambda x : P.\pi$ in the case of →Intro, we can directly tell that it applies. If all rules were syntax directed, it would

```
βℕequal : (ϕ : ctx, T : Type, t1 : T, t2 : T) → option LT(t1 = t2)
βℕequal ϕ T t1 t2 =
  holcase whnf ϕ T t1, whnf ϕ T t2 of
    ((ta : T' → T) tb), (tc td) ↦
      do ⟨pf1⟩ ← βℕequal ϕ (T' → T) ta tc
         ⟨pf1⟩ ← βℕequal ϕ T' tb td
         return ⟨··· proof of ta tb = tc td ···⟩
  | (ta → tb), (tc → td) ↦
      do ⟨pf1⟩ ← βℕequal ϕ Prop ta tc
         ⟨pf1⟩ ← βℕequal ϕ Prop tb td
         return ⟨··· proof of ta → tb = tc → td ···⟩
  | (λx : T.t1), (λx : T.t2) ↦
      do ⟨pf⟩ ← βℕequal [ϕ, x : T] Prop t1 t2
         return ⟨··· proof of λx : T.t1 = λx : T.t2 ···⟩
  | t1, t1 ↦ do return ⟨··· proof of t1 = t1 ···⟩
  | t1, t2 ↦ None

requireEqual : (ϕ : ctx, T : Type, t1 : T, t2 : T).LT(t1 = t2)
requireEqual ϕ T t1 t2 =
  match βℕequal ϕ T t1 t2 with Some x ↦ x | None ↦ error
```

Figure 9. VeriML tactic for checking equality up to β-conversion

be entirely simple to prove that the logic is sound by an inductive argument: essentially, since no proof constructor for False exists, there is no valid derivation for False.

In this logic, the only rule that is not syntax directed is exactly the conversion rule. Therefore, in order to prove the soundness of the logic, we have to show that the conversion rule does not somehow introduce a proof of False. This means that proving the soundness of the logic passes essentially through the specific relation we have chosen for the conversion rule. Therefore, this approach is foundationally limited from supporting user extensions, since any new extension would require a new metatheoretic result in order to make sure that it does not violate logical soundness.

4.2 Throwing conversion away

Since having a fixed conversion rule is bound to fail if we want it to be extensible, what choice are we left with, but to throw it away? This radical sounding approach is what we will do here. We can replace the conversion rule by an explicit notion of equality, and provide explicit proof witnesses for rewriting based on that equality. Essentially, all the points where the conversion rule was alluded to and proofs were omitted, need now be replaced by proof objects witnessing the equivalence. Some details for the additions required to the base λHOL logic are shown in Fig. 8, yielding the λHOL_e logic. There are good reasons for choosing this version: first, the proof checker is as simple as possible, and does not need to include the conversion checking routine. We could view this routine as performing proof search over the replacement rules, so it necessarily is more complicated, especially since it needs to be relatively efficient. Also, the metatheory of the logic itself can be simplified. Even when the conversion rule is supported, the metatheory for the associated logic is proved through the explicit equality approach; this is because model construction for a logic benefits from using explicit equality [Siles and Herbelin 2010].

Still, this approach has a big disadvantage: the proof objects soon become extremely large, since they include painstakingly detailed proofs for even the simplest of equivalences. This precludes their use as independently checkable proof certificates that can be sent to a third party. It is possible that this is one of the reasons why systems based on logics with explicit equalities, such as HOL4

```
whnf : (ϕ : ctx, T : Type, t : T) → (t' : T) × LT(t = t')
whnf ϕ T t = holcase t of
  (t1 : T' → T)(t2 : T') ↦
    let ⟨t1', pf1⟩ = whnf ϕ (T' → T) t1 in
    holcase t1' of
      λx : T'.tf ↦ ⟨[ϕ] tf/[idΦ, t2], ···⟩
    | t1'        ↦ ⟨[ϕ] t1' t2, ···⟩
| natElim_K fz fs n ↦
    let ⟨n', pf1⟩ = whnf ϕ Nat n in holcase n' of
      zero     ↦ ⟨[ϕ] fz, ···⟩
    | succ n'  ↦ ⟨[ϕ] fs n' (natElim_K fz fs n'), ···⟩
    | n'       ↦ ⟨[ϕ] natElim_K fz fs n', ···⟩
| t ↦ ⟨t, ···⟩
```

Figure 10. VeriML tactic for rewriting to weak head-normal form

[Slind and Norrish 2008] and Isabelle/HOL [Nipkow et al. 2002], do not generate proof objects by default.

4.3 Getting conversion back

We will now see how it is possible to reconcile the explicit equality based approach with the conversion rule: we will gain the conversion rule back, albeit it will remain completely outside the logic. Therefore we will be free to extend it, all the while without risking introducing unsoundness in the logic, since the logic remains fixed (λHOL_e as presented above).

We do this by revisiting the view of the conversion rule as a special "trusted" tactic, through the tools presented in the previous section. First, instead of hardcoding a conversion tactic in the type checker, we program a *type-safe conversion tactic*, utilizing the features of VeriML. Based on typing alone we require that it returns a valid proof of the claimed equivalences:

$$\beta\mathbb{N}\text{equal} : (\phi : \text{ctx},\ T : \text{Type},\ t : T,\ t' : T) \to \text{option LT}(t = t')$$

Second, we evaluate this tactic under *proof erasure semantics*. This means that no proof objects are produced, leading to the same space gains as the original conversion rule. Third, we use the staging construct in order to *check conversion statically*.

Details. We now present our approach in more detail. First, in Fig. 9, we show a sketch of the code behind the type-safe conversion check tactic. It works by first rewriting its input terms into weak head-normal form, via the whnf function in Fig. 10, and then recursively checking their subterms for equality. In the equivalence checking function, more cases are needed to deal with quantification; while in the rewriting procedure, a recursive call is missing, which would complicate our presentation here. We also define a version of the tactic that raises an error instead of returning an option type if we fail to prove the terms equal, which we call requireEqual. The full details can be found in our implementation.

The code of the βℕequal tactic is in fact entirely similar to the code one would write for the conversion check routine inside a logic type checker, save for the extra types and proof objects. It therefore follows trivially that everything that holds for the standard implementation of the conversion check also holds for this code: e.g. it corresponds exactly to the $=_{\beta\mathbb{N}}$ relation as defined in the logic; it is bound to terminate because of the strong normalization theorem for this relation; and its proof-erased version is at least as trustworthy as the standard implementation.

Furthermore, given this code, we can produce a form of *typed proof scripts* inside VeriML that correspond exactly to proof objects in the logic with the conversion rule, both in terms of their actual code, and in terms of the steps required to validate them. This is

done by constructing a proof script in VeriML by induction on the derivation of the proof object in λHOL_c, replacing each proof object constructor by an equivalent VeriML tactic as follows:

constructor	to tactic	of type
$\lambda x : P.\pi$	Assume e	$\mathsf{LT}([\phi,\ H : P]\, P') \rightarrow \mathsf{LT}(P \rightarrow P')$
$\pi_1\ \pi_2$	Apply $e_1\ e_2$	$\mathsf{LT}(P \rightarrow P') \rightarrow \mathsf{LT}(P) \rightarrow \mathsf{LT}(P')$
$\lambda x : \mathcal{K}.\pi$	Intro e	$\mathsf{LT}([\phi,\ x : T]\, P') \rightarrow \mathsf{LT}(\forall x : T, P')$
$\pi\ d$	Inst $e\ a$	$\mathsf{LT}(\forall x : T, P) \rightarrow (a : T) \rightarrow \mathsf{LT}(P/[\mathsf{id},\ a])$
c	Lift c	$(H : P) \rightarrow \mathsf{LT}(P)$
(conversion)	Conversion	$\mathsf{LT}(P) \rightarrow \mathsf{LT}(P = P') \rightarrow \mathsf{LT}(P')$

Here we have omitted the current logical environment ϕ; it is maintained through syntactic means as discussed in Sec. 7 and through type inference. The only subtle case is conversion. Given the transformed proof e for the proof object π contained within a use of the conversion rule, we call the conversion tactic as follows:

$$\mathsf{letstatic\ pf\ =\ requireEqual}\ P\ P'\ \mathsf{in\ Conversion\ e\ pf}$$

The arguments to requireEqual can be easily inferred, making crucial use of the rich type information available. Conversion could also be used implicitly in the other tactics. Thus the resulting expression looks entirely identical to the original proof object.

Correspondence with original proof object. In order to elucidate the correspondence between the resulting proof script expression and the original proof object, it is fruitful to view the proof script as a proof certificate, sent to a third party. The steps required to check whether it constitutes a valid proof are the following. First, the whole expression is checked using the type checker of the computational language. Then, the calls to the requireEqual function are evaluated during stage one, using proof erasure semantics. We expect them to be successful, just as we would expect the conversion rule to be applicable when it is used. Last, the rest of the tactics are evaluated; by a simple argument, based on the fact that they do not use pattern matching or side-effects, they are guaranteed to terminate and produce a proof object in λHOL_e. This validity check is entirely equivalent to the behavior of type-checking the λHOL_c proof object, save for pushing all conversion checks towards the end.

4.4 Extending conversion at will

In our treatment of the conversion rule we have so far focused on regaining the $\beta\mathbb{N}$ conversion in our framework. Still, there is nothing confining us to supporting this conversion check only. As long as we can program a conversion tactic in VeriML that has the right type, it can safely be made part of our conversion rule.

For example, we have written an eufEqual function, which checks terms for equivalence based on the equality with uninterpreted functions decision procedure. It is adapted from our previous work on VeriML [Stampoulis and Shao 2010]. This equivalence checking tactic isolates hypotheses of the form $d_1 = d_2$ from the current context, using the newly-introduced context matching support. Then, it constructs a union-find data structure in order to form equivalence classes of terms. Based on this structure, and using code similar to $\beta\mathbb{N}$equal (recursive calls on subterms), we can decide whether two terms are equal up to simple uses of the equality hypotheses at hand. We have combined this tactic with the original $\beta\mathbb{N}$equal tactic, making the implicit equivalence supported similar to the one in the Calculus of Congruent Constructions [Blanqui et al. 2005]. This demonstrates the flexibility of this approach: equivalence checking is extended with a sophisticated decision procedure, which is programmed using its original, imperative formulation. We have programmed both the rewriting procedure and the equality checking procedure in an extensible manner, so that we can globally register further extensions.

4.5 Typed proof scripts as certificates

Earlier we discussed how we can validate the proof scripts resulting from turning the conversion rule into explicit tactic calls. This discussion shows an interesting aspect of typed proof scripts: they can be viewed as a proof witness that is a flexible compromise between untyped proof scripts and proof objects. When a typed proof script consists only of static calls to conversion tactics and uses of total tactics, it can be thought of as a proof object in a logic with the corresponding conversion rule. When it also contains other tactics, that perform potentially expensive proof search, it corresponds more closely to an untyped proof script, since it needs to be fully evaluated. Still, we are allowed to validate parts of it statically. This is especially useful when developing the proof script, because we can avoid the evaluation of expensive tactic calls while we focus on getting the skeleton of the proof correct.

Using proof erasure for evaluating requireEqual is only one of the choices the receiver of such a proof certificate can make. Another choice would be to have the function return an actual proof object, which we can check using the λHOL_e type checker. In that case, the VeriML interpreter does not need to become part of the trusted base of the system. Last, the 'safest possible' choice would be to avoid doing any evaluation of the function, and ask the proof certificate provider to do the evaluation of requireEqual themselves. In that case, no evaluation of computational code would need to happen at the proof certificate receiver's side. This mitigates any concerns one might have for code execution as part of proof validity checking, and guarantees that the small λHOL_e type checker is the trusted base in its entirety. Also, the receiver can decide on the above choices selectively for different conversion tactics – e.g. use proof erasure for $\beta\mathbb{N}$equal but not for eufEqual, leading to a trusted base identical to the λHOL_c case. This means that the choice of the conversion rule rests with the proof certificate receiver and not with the designer of the logic. Thus the proof certificate receiver can choose the level of trust they require at will.

5. Static proof scripts

In the previous section, we have demonstrated how proof checking for typed proof scripts can be made user-extensible, through a new treatment of the conversion rule. It makes use of user-defined, type-safe tactics, which are evaluated statically. The question that remains is what happens with respect to proofs within tactics. If a proof script is found within a tactic, must we wait until that evaluation point is reached to know whether the proof script is correct or not? Or is there a way to check this statically, as soon as the tactic is defined?

In this section we show how this is possible to do in VeriML using the staging construct we have introduced. Still, in this case matters are not as simple as evaluating certain expressions statically rather than dynamically. The reason is that proof scripts contained within tactics mention uninstantiated meta-variables, and thus cannot be evaluated through staging. We resolve this by showing the existence of a transformation, which "collapses" logical terms from an arbitrary meta-variables context into the empty one.

We will focus on the case of developing conversion routines, similar to the ones we saw earlier. The ideas we present are generally applicable when writing other types of tactics as well; we focus on conversion routines in order to demonstrate that the two main ideas we present in this paper can work in tandem.

A rewriter for plus. We will consider the case of writing a rewriter –similar to whnf– for simplifying expressions of the form $x + y$, depending on the second argument. The addition function is defined by induction on the first argument, as follows:

$$(+) = \lambda x.\lambda y.\mathsf{natElim}_{\mathsf{Nat}}\ y\ (\lambda p.\lambda r.\mathsf{Succ}\ r)\ x$$

In order for rewriters to be able to use existing as well as future rewriters to perform their recursive calls, we write them in the open recursion style – they receive a function of the same type that corresponds to the "current" rewriter. The code looks as follows:

```
rewriterType = (ϕ : ctx, T : Type, t : T) → (t' : T) × LT(t = t')
plusRewriter1 : rewriterType → rewriterType
plusRewriter1 recursive ϕ T t = holcase t with
  x + y ↦
    let ⟨y', ⟨pfy'⟩⟩ = recursive ϕ y in
    let ⟨t', ⟨pft'⟩⟩ =
      holcase y' return Σt' : [ϕ] Nat.LT([ϕ] x + y' = t') of
        0       ↦ ⟨x, ··· proof of x + 0 = x ···⟩
      | Succ y' ↦ ⟨ Succ(x + y'),
                    ··· proof of x + Succ y' = Succ (x + y') ··· ⟩
      | y'      ↦ ⟨x + y', ··· proof of x + y' = x + y' ···⟩
    in ⟨t', ⟨··· proof of x + y = t' ···⟩⟩
| t     ↦ ⟨t, ··· proof of t = t ···⟩
```

While developing such a tactic, we can leverage the VeriML type checker to know the types of missing proofs. But how do we fill them in? For the interesting cases of $x+0=x$ and $x+\mathsf{Succ}\ y' = \mathsf{Succ}\ (x+y')$, we would certainly need to prove the corresponding lemmas. But for the rest of the cases, the corresponding lemmas would be uninteresting and tedious to state, such as the following for the $x+y=t'$ case:

$$\mathsf{lemma1} : \forall x, y, y', t', y = y' \rightarrow (x + y' = t') \rightarrow x + y = t$$

Stating and proving such lemmas soon becomes a hindrance when writing tactics. An alternative is to use the congruence closure conversion rule to solve this trivial obligation for us directly at the point where it is required. Our first attempt would be:

```
proof of x + y = t' ≡
  let ⟨pf⟩ = requireEqual [ϕ, H1 : y = y', H2 : x + y' = t'] (x + y) t'
  in ⟨[ϕ] pf/[id_ϕ, pfy', pft']⟩
```

The benefit of this approach is evident when utilizing implicit arguments, since most of the details can be inferred and therefore omitted. Here we had to alter the environment passed to requireEqual, which includes several extra hypotheses. Once the resulting proof has been computed, the hypotheses are substituted by the actual proofs that we have.

The problem with this approach is two-fold: first, the call to the requireEqual tactic is recomputed every time we reach that point of our function. For such a simple tactic call, this does not impact the runtime significantly; still, if we could avoid it, we would be able use more sophisticated and expensive tactics. The second problem is that if for some reason the requireEqual is not able to prove what it is supposed to, we will not know until we actually reach that point in the function.

Moving to static proofs. This is where using the letstatic construct becomes essential. We can evaluate the call to requireEqual statically, during stage one interpretation. Thus we will know at the time that plusRewriter1 is defined whether the call succeeded; also, it will be replaced by a concrete value, so it will not affect the runtime behavior of each invocation of plusRewriter1 anymore. To do that, we need to avoid mentioning any of the metavariables that are bound during runtime, like x, y, and t'. This is done by specifying an appropriate environment in the call to requireEqual, similarly to the way we incorporated the extra knowledge above and substituted it later. Using this approach, we have:

```
proof of x + y = t' ≡
  letstatic ⟨pf⟩ =
    let ϕ' = [x, y, y', t' : Nat, H1 : y = y', H2 : x + y' = t'] in
    requireEqual ϕ' (x + y) t'
  in ⟨[ϕ] pf/[x/id_ϕ, y/id_ϕ, y'/id_ϕ, t'/id_ϕ, pfy'/id_ϕ, pft'/id_ϕ]⟩
```

What we are essentially doing here is replacing the meta-variables by normal logical variables, which our tactics can deal with. The meta-variable context is "collapsed" into a normal context; proofs are constructed using tactics in this environment; last, the resulting proofs are transported back into the desired context by substituting meta-variables for variables. We have explicitly stated the substitutions in order to distinguish between normal logical variables and meta-variables.

The reason why this transformation needs to be done is that functions in our computational language can only manipulate logical terms that are open with respect to a normal variables context; not logical terms that are open with respect to the meta-variables context too. A much more complicated, but also more flexible alternative to using this "collapsing" trick would be to support meta-variables within our computational language directly.

Overall, this approach is entirely similar to proving the auxiliary lemma mentioned above, prior to the tactic definition. The benefit is that by leveraging the type information together with type inference, we can avoid stating such lemmas explicitly, while retaining the same runtime behavior. We thus end up with very concise proof expressions that are statically validated. We introduce syntactic sugar for binding a static proof script to a variable, and then performing a substitution to bring it into the current context, since this is a common operation.

$$\langle e \rangle_{\text{static}} \equiv \mathsf{letstatic}\ \langle \mathsf{pf} \rangle\ =\ e\ \mathsf{in}\ \langle [\phi]\,\mathsf{pf}/\cdots \rangle$$

Based on these, the trivial proofs in the above tactic can be filled in using a simple $\langle \mathsf{requireEqual} \rangle_{\text{static}}$ call; for the other two we use $\langle \mathsf{Instantiate}\ (\mathsf{NatInduction}\ \mathsf{requireEqual}\ \mathsf{requireEqual})\ x \rangle_{\text{static}}$.

After we define plusRewriter1, we can register it with the global equivalence checking procedure. Thus, all later calls to requireEqual will benefit from this simplification. It is then simple to prove commutativity for addition:

```
plusComm : LT(∀x, y. x + y = y + x)
plusComm = NatInduction requireEqual requireEqual
```

Based on this proof, we can write a rewriter that takes commutativity into account and uses the hash values of logical terms to avoid infinite loops. We have worked on an arithmetic simplification rewriter that is built by layering such rewriters together, using previous ones to aid us in constructing the proofs required in later ones. It works by converting expressions into a list of monomials, sorting the list based on the hash values of the variables, and then factoring monomials on the same variable. Also, the eufEqual procedure mentioned earlier has all of its associated proofs automated through static proof scripts, using a naive, potentially non-terminating, equality rewriter.

Is collapsing always possible? A natural question to ask is whether collapsing the metavariables context into a normal context is always possible. In order to cast this as a more formal question, we notice that the essential step is replacing a proof object π of type $[\Phi]t$, typed under the meta-variables environment Ψ, by a proof object π' of type $[\Phi']t'$ typed under the empty meta-variables environment. There needs to be a substitution so that π' gets transported back to the Φ, Ψ environment, and has the appropriate type.

Syntax of the logic

(terms) $t ::= s \mid c \mid f_i \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \mid \mathsf{refl}\ t \mid \mathsf{leibniz}\ t_1\ t_2 \mid \mathsf{lamEq}\ t \mid \mathsf{forallEq}\ t_1\ t_2 \mid \mathsf{betaEq}\ t_1\ t_2$
(sorts) $s ::= \mathsf{Prop} \mid \mathsf{Type} \mid \mathsf{Type}'$ *(var. context)* $\Phi ::= \bullet \mid \Phi,\ t$ *(substitutions)* $\sigma ::= \bullet \mid \sigma,\ t$

Example of representation: $a : \mathsf{Nat} \vdash \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.\mathsf{refl}\ (\mathsf{plus}\ a\ y))(\mathsf{plus}\ a\ x) \mapsto \mathsf{Nat} \vdash \lambda(\mathsf{Nat}).(\lambda(\mathsf{Nat}).\mathsf{refl}\ (\mathsf{plus}\ f_0\ b_0))\ (\mathsf{plus}\ f_0\ b_0)$

Freshen: $\lceil t \rceil^n_m$

$$\begin{array}{lcl} \lceil f_i \rceil & = & f_i \\ \lceil b_n \rceil^n_m & = & f_m \\ \lceil b_i \rceil^n & = & b_i \text{ when } i < n \\ \lceil (\lambda(t_1).t_2) \rceil^n & = & \lambda(\lceil t_1 \rceil^n).\lceil t_2 \rceil^{n+1} \\ \lceil t_1\ t_2 \rceil & = & \lceil t_1 \rceil\ \lceil t_2 \rceil \end{array}$$

Bind: $\lfloor t \rfloor^n_m$

$$\begin{array}{lcl} \lfloor f_{m-1} \rfloor^n_m & = & b_n \\ \lfloor f_i \rfloor^n_m & = & f_i \text{ when } i < m-1 \\ \lfloor b_i \rfloor & = & b_{i+1} \\ \lfloor (\lambda(t_1).t_2) \rfloor & = & \lambda(\lfloor t_1 \rfloor^n).\lfloor t_2 \rfloor^{n+1} \\ \lfloor t_1\ t_2 \rfloor & = & \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \end{array}$$

(a) Hybrid deBruijn levels-deBruijn indices representation technique

---

Syntax $t ::= \cdots \mid f_{\mathbf{I}} \mid X_i/\sigma$ $\Phi ::= \bullet \mid \Phi,\ t \mid \Phi,\ \phi_i$ $\sigma ::= \bullet \mid \sigma,\ t \mid \sigma,\ \mathbf{id}(\phi_i)$ *(indices)* $\mathbf{I} ::= n \mid \mathbf{I} + |\phi_i|$ *(ctx.terms)* $T ::= [\Phi]t \mid [\Phi]\Phi'$
(ctx.kinds) $K ::= [\Phi]t \mid [\Phi]\,\mathsf{ctx}$ *(extension context)* $\Psi ::= \bullet \mid \Psi,\ K$ *(ext. subst.)* $\sigma_\Psi ::= \bullet \mid \sigma_\Psi,\ T$

$\Psi;\ \Phi \vdash t : t'$ (sample)

$$\frac{\Phi.\mathbf{I} = t}{\Psi;\ \Phi \vdash f_{\mathbf{I}} : t} \qquad \frac{\Psi;\ \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\ \Phi \vdash t_2 : t}{\Psi;\ \Phi \vdash t_1\ t_2 : \lceil t' \rceil \cdot (\mathsf{id}_\Phi, t_2)} \qquad \frac{\Psi.i = [\Phi']t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}$$

$\Psi \vdash T : K$

$$\frac{\Psi;\ \Phi \vdash t : t'}{\Psi \vdash [\Phi]t : [\Phi]t'} \qquad \frac{\Psi \vdash \Phi,\ \Phi'\ \mathsf{wf}}{\Psi \vdash [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}}$$

$\Psi \vdash \Phi$ wf (sample)

$$\frac{\Psi \vdash \Phi\ \mathsf{wf} \qquad \Psi.i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash (\Phi,\ \phi_i)\ \mathsf{wf}}$$

(b) Extension variables: meta-variables and context variables

---

Subst. application: $t \cdot \sigma$ $\quad c \cdot \sigma = c \quad f_{\mathbf{I}} \cdot \sigma = \sigma.\mathbf{I} \quad b_i \cdot \sigma = b_i \quad (\lambda(t_1).t_2) \cdot \sigma = \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) \quad (t_1\ t_2) \cdot \sigma = (t_1 \cdot \sigma)\ (t_2 \cdot \sigma)$

Ext. subst. application (sample)

$$(\mathbf{I},\ |\phi_i|) \cdot \sigma_\Psi = (\mathbf{I} \cdot \sigma_\Psi),\ |\Phi'| \text{ when } \sigma_\Psi.i = [\_]\Phi' \qquad (X_i/\sigma) \cdot \sigma_\Psi = t \cdot (\sigma \cdot \sigma_\Psi) \text{ when } \sigma_\Psi.i = [\_]t$$
$$(\sigma,\ \mathbf{id}(\phi_i)) \cdot \sigma_\Psi = \sigma \cdot \sigma_\Psi,\ \mathsf{id}_{\sigma_\Psi.i} \qquad (\Phi,\ \phi_i) \cdot \sigma_\Psi = \Phi \cdot \sigma_\Psi,\ \Phi' \text{ when } \sigma_\Psi.i = [\_]\Phi'$$

$\Psi;\ \Phi \vdash \sigma : \Phi'$

$$\frac{}{\Psi;\ \Phi \vdash \bullet : \bullet} \qquad \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi;\ \Phi \vdash t : t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) : (\Phi',\ t')} \qquad \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\ \phi_i \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi',\ \phi_i)}$$

$\Psi \vdash \sigma_\Psi : \Psi'$ (selected)

$$\frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : K \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\ T) : (\Psi',\ K)}$$

Subst. lemmas:

$$\frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi' \vdash \sigma : \Phi}{\Psi;\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma} \qquad \frac{\Psi;\ \Phi' \vdash \sigma : \Phi \qquad \Psi;\ \Phi'' \vdash \sigma' : \Phi'}{\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi} \qquad \frac{\Psi \vdash T : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi}$$

(c) Substitutions over logical variables and extension variables

---

Syntax: $\Gamma ::= \bullet \mid \Gamma,\ x : \tau \mid \Gamma,\ x :_s \tau \mid \Gamma,\ \alpha : k$ $\qquad e ::= \cdots \mid \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e'$

Limit ctx:

$$\begin{array}{lcl} \bullet|_{\text{static}} & = & \bullet \\ (\Gamma,\ x :_s t)|_{\text{static}} & = & \Gamma|_{\text{static}},\ x : t \\ (\Gamma,\ x : t)|_{\text{static}} & = & \Gamma|_{\text{static}} \\ (\Gamma,\ \alpha : k)|_{\text{static}} & = & \Gamma|_{\text{static}} \end{array}$$

$\Psi;\ \Sigma;\ \Gamma \vdash e : \tau$ (part)

$$\frac{\bullet;\ \Sigma;\ \Gamma|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \Sigma;\ \Gamma, x :_s \tau \vdash e' : \tau}{\Psi;\ \Sigma;\ \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau} \qquad \frac{x :_s \tau \in \Gamma}{\Psi;\ \Sigma;\ \Gamma \vdash x : \tau}$$

Evaluation:

$v ::= \Lambda(K).e_d \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ v \mid () \mid \lambda x : \tau.e_d \mid (v,\ v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e_d$
$\mathcal{S} ::= \mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e' \mid \mathsf{letstatic}\ x\ =\ \mathcal{S}\ \mathsf{in}\ e' \mid \Lambda(K).\mathcal{S} \mid \lambda x : \tau.\mathcal{S} \mid \mathsf{unpack}\ e_d\ (.)x.(\mathcal{S}) \mid \mathsf{case}(e_d, x.\mathcal{S}, x.e_2)$
$\quad \mid \mathsf{case}(e_d, x.e_d, x.\mathcal{S}) \mid \Lambda\alpha : k.\mathcal{S} \mid \mathsf{fix}\ x : \tau.\mathcal{S} \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto \mathcal{S}) \mid \mathcal{E}_s[\mathcal{S}]$
$\mathcal{E}_s ::= \mathcal{E}_s\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ \mathcal{E}_s \mid \mathsf{unpack}\ \mathcal{E}_s\ (.)x.(e') \mid \mathcal{E}_s\ e' \mid e_d\ \mathcal{E}_s \mid (\mathcal{E}_s,\ e) \mid (e_d,\ \mathcal{E}_s) \mid \mathsf{proj}_i\ \mathcal{E}_s \mid \mathsf{inj}_i\ \mathcal{E}_s$
$\quad \mid \mathsf{case}(\mathcal{E}_s, x.e_1, x.e_2) \mid \mathsf{fold}\ \mathcal{E}_s \mid \mathsf{unfold}\ \mathcal{E}_s \mid \mathsf{ref}\ \mathcal{E}_s \mid \mathcal{E}_s := e' \mid e_d := \mathcal{E}_s \mid !\mathcal{E}_s \mid \mathcal{E}_s\ \tau$
$e_d ::=$ all of e except $\mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e'$ $\qquad \mathcal{E} ::=$ exactly as $\mathcal{E}_s$ with $\mathcal{E}_s \to \mathcal{E}$ and $e \to e_d$

Stage 1 op.sem.:

$$\frac{(\ \mu,\ e_d\ ) \longrightarrow (\ \mu',\ e'_d\ )}{(\ \mu,\ \mathcal{S}[e_d]\ ) \longrightarrow_s (\ \mu',\ \mathcal{S}[e'_d]\ )} \qquad (\ \mu,\ \mathcal{S}[\mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e]\ ) \longrightarrow_s (\ \mu,\ \mathcal{S}[e[v/x]]\ )$$

$$(\ \mu,\ \mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e\ ) \longrightarrow_s (\ \mu,\ e[v/x]\ )$$

(d) Computational language: staging support

Figure 11. Main definitions in metatheory

We have proved that this is possible under certain restrictions: the types of the metavariables in the current context need to depend on the same free variables context $\Phi_{\max}$, or prefixes of that context. Also the substitutions they are used with need to be prefixes of the identity substitution for $\Phi_{\max}$. Such terms are characterized as collapsible. We have proved that collapsible terms can be replaced using terms that do not make use of metavariables; more details can be found in Sec. 6 and in the accompanying technical report [Stampoulis and Shao 2012].

This restriction corresponds very well to the treatment of variable contexts in the Delphin language. This language assumes an ambient context of logical variables, instead of full, contextual modal terms. Constructs to extend this context and substitute a specific variable exist. If this last feature is not used, the ambient context grows monotonically and the mentioned restriction holds trivially. In our tests, this restriction has not turned out to be limiting.

6. Metatheory

We have completed an extensive reworking of the metatheory of VeriML, in order to incorporate the features that we have presented in this paper. Our new metatheory includes a number of technical advances compared to our earlier work [Stampoulis and Shao 2010]. We will present a technical overview of our metatheory in this section; full details can be found in our technical report [Stampoulis and Shao 2012].

Variable representation technique. Though our metatheory is done on paper, we have found that using a concrete variable representation technique elucidates some aspects of how different kinds of substitutions work in our language, compared to having normal named variables. For example, instantiating a context variable with a concrete context triggers a set of potentially complicated α-renamings, which a concrete representation makes explicit. We use a hybrid technique representing bound variables as deBruijn indices, and free variables as deBruijn levels. Our technique is a small departure from the named approach, requiring fewer extra annotations and lemmas than normal deBruijn indices. Also it identifies terms not only up to α-equivalence, but also up to extension of the context with new variables; this is why it is also used within the VeriML implementation.The two fundamental operations of this technique are freshening and binding, which are shown in Fig. 11a.

Extension variables. We extend the logic with support for metavariables and context variables – we refer to both these sorts of variables as extension variables. A meta-variable X_i stands for a contextual term $T = [\Phi]t$, which packages a term together with the context it inhabits. Context variables ϕ_i stand for a context Φ, and are used to "weaken" parametric contexts in specific positions. Both kinds of variables are needed to support manipulation of open logical terms. Details of their definition and typing are shown in Fig. 11b. We use the same hybrid approach as above for representing these variables. A somewhat subtle aspect of this extension is that we generalize the deBruijn levels $\mathbf{I}$ used to index free variables, in order to deal effectively with parametric contexts.

Substitutions. The hybrid representation technique we use for variables renders simultaneous substitutions for all variables in scope as the most natural choice. In Fig. 11c, we show some example rules of how to apply a full simultaneous substitution σ to a term t, denoted as $t \cdot \sigma$. Similarly, we define full simultaneous substitutions σ_Ψ for extension contexts; defining their application has a very natural description, because of our variable representation technique. We prove a number of substitution lemmas which have simple statements, as shown in Fig. 11c. The proofs of these lemmas comprise the main effort required in proving the type-safety of a computational language such as the one we support, as they represent the point where computation specific to logical term manipulation takes place.

Computational language. We define an ML-style computational language that supports dependent functions and dependent pairs over contextual terms T, as well as pattern matching over them. Lack of space precludes us from including details here; full details can be found in the accompanying technical report [Stampoulis and Shao 2012]. A fairly complete ML calculus is supported, with mutable references and recursive types. Type safety is proved using standard techniques; its central point is extending the logic substitution lemmas to expressions and using them to prove progress and preservation of dependent functions and dependent pairs. This proof is modular with respect to the logic and other logics can easily be supported.

Pattern matching. Our metatheory includes many extensions in the pattern matching that is supported, as well as a new approach for dealing with typing patterns. We include support for pattern matching over contexts (e.g. to pick out hypotheses from the context) and for non-linear patterns. The allowed patterns are checked through a restriction of the usual typing rules $\Psi \vdash_p T : K$.

The essential idea behind our approach to pattern matching is to identify what the relevant variables in a typing derivation are. Since contexts are ordered, "removing" non-relevant variables amounts to replacing their definitions in the context with holes, which leads us to partial contexts $\widehat{\Psi}$. The corresponding notion of partial substitutions is denoted as $\widehat{\sigma_\Psi}$. Our main theorem about pattern matching can then be stated as:

Theorem 6.1 (Decidability of pattern matching) *If $\Psi \vdash_p T : K$, $\bullet \vdash_p T' : K$ and $\mathsf{relevant}(\Psi; \Phi \vdash T : K) = \widehat{\Psi}$, then either there exists a unique partial substitution $\widehat{\sigma_\Psi}$ such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$ and $T \cdot \widehat{\sigma_\Psi} = T'$, or no such substitution exists.*

Staging. Our development in this paper critically depends on the $\mathsf{letstatic}$ construct we presented earlier. It can be seen as a dual of the traditional box construct of Davies and Pfenning [1996]. Details of its typing and semantics are shown in Fig. 11d. We define a notion of "static evaluation contexts" $\mathcal{S}$, which enclose a hole of the form $\mathsf{letstatic}\, x = \bullet\, \mathsf{in}\, e$. They include normal evaluation contexts, as well as evaluation contexts under binding structures. We evaluate expressions e that include staging constructs using the $\longrightarrow_s$ relation; internally, this uses the normal evaluation rules, that are used in the second stage as well, for evaluating expressions which do not include other staging constructs. If stage-one evaluation is successful, we are left with a residual dynamic configuration (μ', e_d) which is then evaluated normally. We prove type-safety for stage-one evaluation; its statement follows.

Theorem 6.2 (Stage-one Type Safety) *If $\bullet; \Sigma; \bullet \vdash e : \tau$ then: either e is a dynamic expression e_d; or, for every store μ such that $\vdash \mu : \Sigma$, we have: either $\mu, e \longrightarrow_s \mathsf{error}$, or, there exists an e', a new store typing $\Sigma' \supseteq \Sigma$ and a new store μ' such that: $(\mu, e) \longrightarrow (\mu', e')$; $\vdash \mu' : \Sigma'$; and $\bullet; \Sigma'; \bullet \vdash e' : \tau$.*

Collapsing extension variables. Last, we have proved the fact that under the conditions described in Sec. 5, it is possible to collapse a term t into a term t' which is typed under the empty extension variables context; a substitution σ with which we can regain the original term t exists. This suggests that whenever a proof object t for a specific proposition is required, an equivalent proof object that does not mention uninstantiated extension variables exists. Therefore, we can write an equivalent proof script producing the collapsed proof object instead, and evaluate that script statically. The statement of this theorem is the following:

Theorem 6.3 *If $\Psi \vdash [\Phi]t : [\Phi]t_T$ and $\mathsf{collapsible}(\Psi \vdash [\Phi]t : [\Phi]t_T)$, then there exist Φ', t', t'_T and σ such that $\bullet \vdash \Phi'$ wf, $\bullet \vdash [\Phi']t' : [\Phi']t'_T$, $\Psi; \Phi \vdash \sigma : \Phi'$, $t' \cdot \sigma = t$ and $t'_T \cdot \sigma = t_T$.*

The main idea behind the proof is to maintain a number of substitutions and their inverses: one to go from a general Ψ extension context into an "equivalent" Ψ' context, which includes only definitions of the form $[\Phi]t$, for a constant Φ context that uses no extension variables. Then, another substitution and its inverse are maintained to go from that extension variables context into the empty one; this is simpler, since terms typed under Ψ' are already essentially free of metavariables. The computational content within the proof amounts to a procedure for transforming proof scripts inside tactics into static proof scripts.

7. Implementation

We have completed a prototype implementation of the VeriML language, as described in this paper, that supports all of our claims. We have built on our existing prototype [Stampoulis and Shao 2010] and have added an extensive set of new features and improvements. The prototype is written in OCaml and is about 6k lines of code. Using the prototype we have implemented a number of examples, that are about 1.5k lines of code. Readers are encouraged to download and try the prototype from `http://flint.cs.yale.edu/publications/supc.html`.

New features. We have implemented the new features we have described so far: context matching, non-linear patterns, proof-erasure semantics, staging, and inferencing for logical and computational terms. Proof-erasure semantics are utilized only if requested by a per-function flag, enabling us to selectively "trust" tactics. The staging construct we support is more akin to the $\langle\cdot\rangle_{\text{static}}$ form described as syntactic sugar in Sec. 5, and it is able to infer the collapsing substitutions that are needed, following the approach used in our metatheory.

Changes. We have also changed quite a number of things in the prototype and improved many of its aspects. A central change, mediated by our new treatment of the conversion rule, was to modify the used logic in order to use the explicit equality approach; the existing prototype used the λHOL_c logic. We also switched the variable representation to the hybrid deBruijn levels-deBruijn indices technique we described, which enabled us to implement subtyping based on context subsumption. Also, we have adapted the typing rules of the pattern matching construct in order to support refining the environment based on the current branch.

Examples implemented. We have implemented a number of examples to support our claims. First, we have written the type-safe conversion check routine for $\beta\mathbb{N}$, and extended it to support congruence closure based on equalities in the context. Proofs of this latter tactic are constructed automatically through static proof scripts, using a naive rewriter that is non-terminating in the general case. We have also completed proofs for theorems of arithmetic for the properties of addition and multiplication, and used them to write an arithmetic simplification tactic. All of the theorems are proved by making essential use of existing conversion rules, and are immediately added into new conversion rules, leading to a compact and clean development style. The resulting code does not need to make use of translation validation or proof by reflection, which are typically used to implement similar tactics in existing proof assistants.

Towards a practical proof assistant. In order to facilitate practical proof and program construction in VeriML, we introduced some features to support surface syntax, enabling users to omit most details about the environments of contextual terms and the substitutions used with meta-variables. This syntax follows the style of Delphin, assuming an ambient logical variable environment which is extended through a construct denoted as $\nu x : t.e$. Still, the full power of contextual modal type theory is available, which is crucial in order to change what the current ambient environment is, used, as we saw earlier, for static calls to tactics. In general the surface syntax leads to much more concise and readable code.

Last, we introduced syntax support for calls to tactics, enabling users to write proof expressions that look very similar to proof scripts in current proof assistants. We developed a rudimentary ProofGeneral mode for VeriML, that enables us to call the VeriML type-checker and interpreter for parts of source files. By adding holes to our sources, we can be informed by the type inference mechanism about their expected types. Those types correspond to what the current "proof state" is at that point. Therefore, a possible workflow for developing tactics or proofs, is writing the known parts, inserting holes in missing points to know what remains to be proved, and calling the typechecker to get the proof state information. This workflow corresponds closely to the interactive proof development support in proof assistants like Coq and Isabelle, but generalizes it to the case of tactics as well.

8. Related work

There is a large body of work that is related to the ideas we have presented here.

Techniques for robust proof development. There have been multiple proposals for making proof development inside existing proof assistants more robust. A well-known technique is *proof-by-reflection* [Boutin 1997]: writing total and certified decision procedures within the functional language contained in a logic like CIC. A recently introduced technique is *automation through canonical structures* [Gonthier et al. 2011]: the resolution mechanism for finding instances of canonical structures (a generalization of type classes) is cleverly utilized in order to program automation procedures for specific classes of propositions. We view both approaches as somewhat similar, as both are based in cleverly exploiting static "interpreters" that are available in a modern proof assistant: the partial evaluator within the conversion rule in the former case; the unification algorithm within instance discovery in the latter case.

Our approach can thus be seen as similar, but also as a generalization of these approaches, since a general-purpose programming model is supported. Therefore, users do not have to adapt to a specific programming style for writing automation code, but can rather use a familiar functional language. Proof-by-reflection could perhaps be used to support the same kind of extensions to the conversion rule; still, this would require reflecting a large part of the logic in itself, through a prohibitively complicated encoding. Both techniques are applicable to our setting as well and could be used to provide benefits to large developments within our language.

The style advocated in Chlipala [2011] (and elsewhere) suggests that proper proof engineering entails developing sophisticated automation tactics in a modular style, and extending their power by adding proved lemmas as hints. We are largely inspired by this approach, and believe that our introduction of the extensible conversion rule and static checking of tactics can significantly benefit it. We demonstrate similar ideas in layering conversion tactics.

Traditional proof assistants. There are many parallels of our work with the *LCF family of proof assistants*, like HOL4 [Slind and Norrish 2008] and HOL-Light [Harrison 1996], which have served as inspiration. First, the foundational logic that we use is similar. Also, our use of a dedicated ML-like programming language to program tactics and proof scripts is similar to the approach taken by HOL4 and HOL-Light. Last, the fact that no proof objects need to be generated is shared. Still, checking a proof script in HOL requires evaluating it fully. Using our approach, we can selectively evaluate parts of proof scripts; we focus on conversion-like tactics, but we are not limited inherrently to those. This is only possible because our proof scripts carry proof state information within their types. Similarly, proof scripts contained within LCF tactics cannot

be evaluated statically, so it is impossible to establish their validity upon tactic definition. It is possible to do a transformation similar to ours manually (lifting proof scripts into auxiliary lemmas that are proved prior to the tactic), but the lack of type information means that many more details need to be provided.

The Coq proof assistant [Barras et al. 2010] is another obvious point of reference for our work. We will focus on the conversion rule that CIC, its accompanying logic, supports – the same problems with respect to proof scripts and tactics that we described in the LCF case also apply for Coq. The conversion rule, which identifies computationally equivalent propositions, coupled with the rich type universe available, opens up many possibilities for constructing small and efficiently checkable proof objects. The implementation of the conversion rule needs to be part of the trusted base of the proof assistant. Also, the fact that the conversion check is built-in to the proof assistant makes the supported equivalence rigid and non-extensible by frequently used decision procedures.

There is a large body of work that aims to extend the conversion rule to arbitrary confluent rewrite systems (e.g. Blanqui et al. [1999]) and to include decision procedures [Strub 2010]. These approaches assume some small or larger addition to the trusted base, and extend the already complex metatheory of Coq. Furthermore, the NuPRL proof assistant [Constable et al. 1986] is based on extensional type theory which includes an extensional conversion rule. This enables complex decision procedures to be part of conversion; but it results in a very large trusted base. We show how, for a subset of these type theories, the conversion check can be recovered outside the trusted base. It can be extended with arbitrarily complex new tactics, written in a familiar programming style, without any metatheoretic additions and without hurting the soundness of the logic. The question of whether these type theories can be supported in full remains as future work, but as far as we know, there is no inherrent limitation to our approach.

Dependently-typed programming. The large body of work on dependently-typed languages has close parallels to our work. Out of the multitude of proposals, we consider the Russell framework [Sozeau 2006] as the current state-of-the-art, because of its high expressivity and automation in discharging proof obligations. In our setting, we can view dependently-typed programming as a specific case of tactics producing complex data types that include proof objects. Static proof scripts can be leveraged to support expressivity similar to the Russell framework. Furthermore, our approach opens up a new intriguing possibility: dependently-typed programs whose obligations are discharged statically and automatically, through code written within the same language.

Last, we have been largely inspired by the work on languages like Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008], and build on our previous work on VeriML [Stampoulis and Shao 2010]. We investigate how to leverage type-safe tactics, as well as a number of new constructs we introduce, so as to offer an extensible notion of proof checking. Also, we address the issue of statically checking the proof scripts contained within tactics written in VeriML. As far as we know, our development is the first time languages such as these have been demonstrated to provide a workflow similar to interactive proof assistants.

Acknowledgments

We thank anonymous referees for their suggestions and comments on an earlier version of this paper. This research is based on work supported in part by DARPA CRASH grant FA8750-10-2-0254 and NSF grants CCF-0811665, CNS-0910670, and CNS 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

H.P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual (version 8.3), 2010.

F. Blanqui, J.P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *Rewriting Techniques and Applications*, pages 671–671. Springer, 1999.

F. Blanqui, J.P. Jouannaud, and P.Y. Strub. A calculus of congruent constructions. *Unpublished draft*, 2005.

S. Boutin. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science*, 1281:515–529, 1997.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2011.

R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270. ACM, 1996.

G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55 (11):1382–1393, 2008.

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 163–175. ACM, 2011.

J. Harrison. HOL Light: A tutorial introduction. *Lecture Notes in Computer Science*, pages 265–269, 1996.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

T. Nipkow, L.C. Paulson, and M. Wenzel. Isabelle/HOL : A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS, 2002.

B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 163–173. ACM New York, NY, USA, 2008.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. *Lecture Notes in Computer Science*, 4960:93, 2008.

V. Siles and H. Herbelin. Equality is typable in semi-full pure type systems. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 21–30. IEEE, 2010.

K. Slind and M. Norrish. A brief overview of HOL4. *Theorem Proving in Higher Order Logics*, pages 28–32, 2008.

M. Sozeau. Subset coercions in coq. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, pages 237–252. Springer-Verlag, 2006.

A. Stampoulis and Z. Shao. VeriML: Typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 333–344. ACM, 2010.

A. Stampoulis and Z. Shao. Static and user-extensible proof checking (extended version). Available in the ACM Digital Library, 2012.

P.Y. Strub. Coq modulo theory. In *Proceedings of the 24th International Conference on Computer Science Logic*, pages 529–543. Springer-Verlag, 2010.

Static and user-extensible proof checking

Extended Version

Antonis Stampoulis Zhong Shao

Department of Computer Science
Yale University
New Haven, CT 06520, USA

{antonis.stampoulis,zhong.shao}@yale.edu

Abstract

Despite recent successes, large-scale proof development within proof assistants remains an arcane art that is extremely time-consuming. We argue that this can be attributed to two profound shortcomings in the architecture of modern proof assistants. The first is that proofs need to include a large amount of minute detail; this is due to the rigidity of the proof checking process, which cannot be extended with domain-specific knowledge. In order to avoid these details, we rely on developing and using tactics, specialized procedures that produce proofs. Unfortunately, tactics are both hard to write and hard to use, revealing the second shortcoming of modern proof assistants. This is because there is no static knowledge about their expected use and behavior.

As has recently been demonstrated, languages that allow type-safe manipulation of proofs, like Beluga, Delphin and VeriML, can be used to partly mitigate this second issue, by assigning rich types to tactics. Still, the architectural issues remain. In this paper, we build on this existing work, and demonstrate two novel ideas: an *extensible conversion rule* and support for *static proof scripts*. Together, these ideas enable us to support both user-extensible proof checking, and sophisticated static checking of tactics, leading to a new point in the design space of future proof assistants. Both ideas are based on the interplay between a light-weight staging construct and the rich type information available.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Verification

1. Introduction

There have been various recent successes in using proof assistants to construct foundational proofs of large software, like a C compiler [Leroy 2009] and an OS microkernel [Klein et al. 2009], as well as complicated mathematical proofs [Gonthier 2008]. Despite this success, the process of large-scale proof development using the foundational approach remains a complicated endeavor that requires significant manual effort and is plagued by various architectural issues.

The big benefit of using a foundational proof assistant is that the proofs involved can be checked for validity using a very small proof checking procedure. The downside is that these proofs are very large, since proof checking is fixed. There is no way to add domain-specific knowledge to the proof checker, which would enable proofs that spell out less details. There is good reason for this, too: if we allowed arbitrary extensions of the proof checker, we could very easily permit it to accept invalid proofs.

Because of this lack of extensibility in the proof checker, users rely on tactics: procedures that produce proofs. Users are free to write their own tactics, that can create domain-specific proofs. In fact, developing domain-

specific tactics is considered to be good engineering when doing large developments, leading to significantly decreased overall effort – as shown, e.g. in Chlipala [2011]. Still, using and developing tactics is error-prone. Tactics are essentially untyped functions that manipulate logical terms, and thus tactic programming is untyped. This means that common errors, like passing the wrong argument, or expecting the wrong result, are not caught statically. Exacerbating this, proofs contained within tactics are not checked statically, when the tactic is defined. Therefore, even if the tactic is used correctly, it could contain serious bugs that manifest only under some conditions.

With the recent advent of programming languages that support strongly typed manipulation of logical terms, such as Beluga [Pientka and Dunfield 2008], Delphin [Poswolsky and Schürmann 2008] and VeriML [Stampoulis and Shao 2010], this situation can be somewhat mitigated. It has been shown in Stampoulis and Shao [2010] that we can specify what kinds of arguments a tactic expects and what kind of proof it produces, leading to a type-safe programming style. Still, this does not address the fundamental problem of proof checking being fixed – users still have to rely on using tactics. Furthermore, the proofs contained within the type-safe tactics are in fact proof-producing programs, which need to be evaluated upon invocation of the tactic. Therefore proofs within tactics are not checked statically, and they can still cause the tactics to fail upon invocation.

In this paper, we build on the past work on these languages, aiming to solve both of these issues regarding the architecture of modern proof assistants. We introduce two novel ideas: support for an **extensible conversion rule** and **static proof scripts** inside tactics. The former technique enables proof checking to become user-extensible, while maintaining the guarantee that only logically sound proofs are admitted. The latter technique allows for statically checking the proofs contained within tactics, leading to increased guarantees about their runtime behavior. Both techniques are based on the same mechanism, which consists of a light-weight staging construct. There is also a deep synergy between them, allowing us to use the one to the benefit of the other.

Our main contributions are the following:

- First, we present what we believe is the first technique for having an extensible conversion rule, which combines the following characteristics: it is safe, meaning that it preserves logical soundness; it is user-extensible, using a familiar, generic programming model; and, it does not require metatheoretic additions to the logic, but can be used to simplify the logic instead.
- Second, building on existing work for typed tactic development, we introduce static checking of the proof scripts contained within tactics. This significantly reduces the development effort required, allowing us to write tactics that benefit from existing tactics and from the rich type information available.
- Third, we show how typed proof scripts can be seen as an alternative form of proof witness, which falls between a proof object and a proof script. Receivers of the certificate are able to decide on the tradeoff between the level of trust they show and the amount of resources needed to check its validity.

In terms of technical contributions, we present a number of technical advances in the metatheory of the aforementioned programming languages. These include a simple staging construct that is crucial to our development and a new technique for variable representation. We also show a condition under which static checking of proof scripts inside tactics is possible. Last, we have extended an existing prototype implementation with a significant number of features, enabling it to support our claims, while also rendering its use as a proof assistant more practical.

2. Informal presentation

Glossary of terms. We will start off by introducing some concepts that will be used throughout the paper. The first fundamental concept we will consider is the notion of a *proof object*: given a derivation of a proposition inside a formal logic, a proof object is a term representation of this derivation. A *proof checker* is a program that can decide whether a given proof object is a valid derivation of a specific proposition or not. Proof objects are extremely verbose and are thus hard to write by hand. For this reason, we use *tactics*: functions that produce

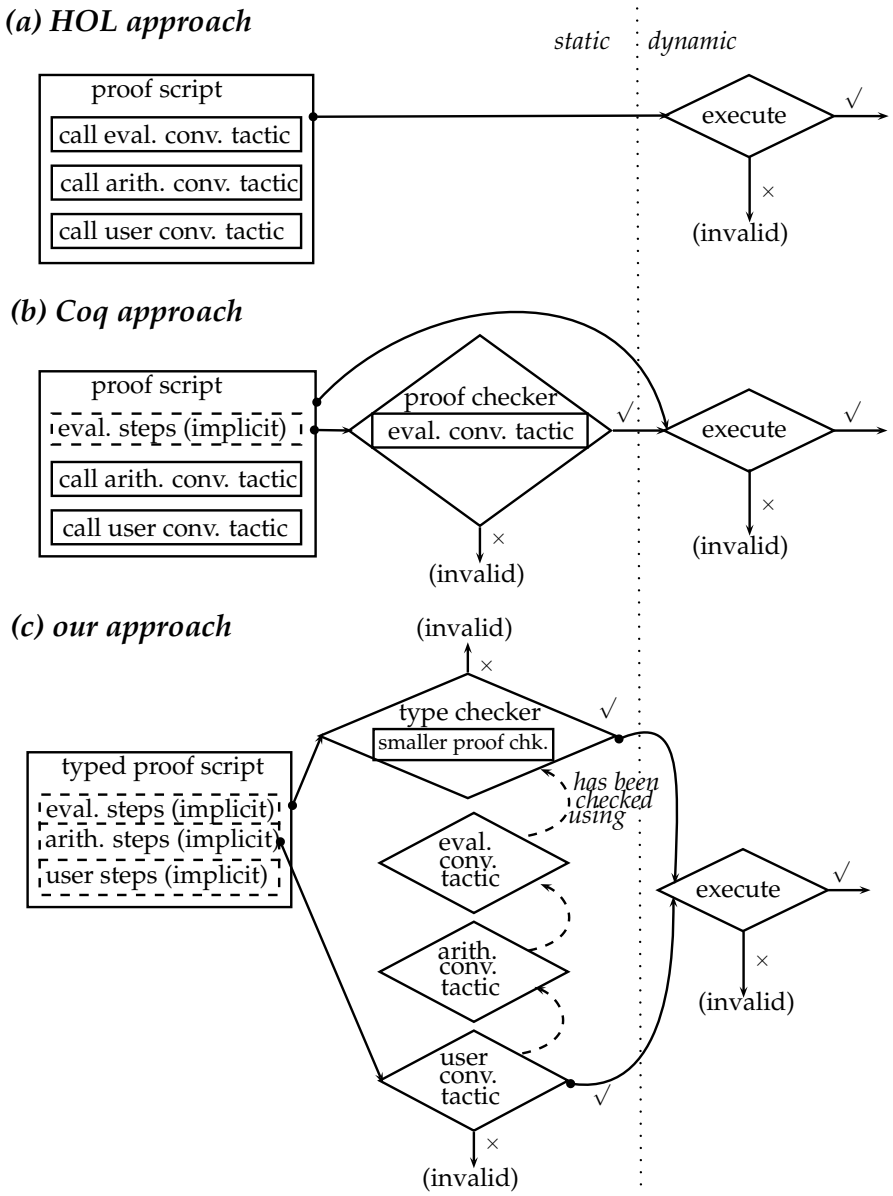


Figure 1. Checking proof scripts in various proof assistants

proof objects. By combining tactics together, we create proof-producing programs, which we call *proof scripts*. If a proof script is evaluated, and the evaluation completes successfully, the resulting proof object can be checked using the original proof checker. In this way, the trusted base of the system is kept at the absolute minimum. The language environment where proof scripts and tactics are written and evaluated is called a *proof assistant*; evidently, it needs to include a proof checker.

Checking proof objects. In order to keep the size of proof objects manageable, many of the logics used for mechanized proof checking include a *conversion rule*. This rule is used implicitly by the proof checker to decide whether any two propositions are equivalent; if it determines that they are indeed so, the proof of their equivalence can be omitted. We can thus think of it as a special tactic that is embedded within the proof checker, and used implicitly.

The more sophisticated the relation supported by the conversion rule is, the simpler are proof objects to write, since more details can be omitted. On the other hand, the proof checker becomes more complicated, as does the metatheory proof showing the soundness of the associated logic. The choice in Coq [Barras et al. 2010], one of the most widely used proof assistants, with respect to this trade-off, is to have a conversion rule that identifies propositions up to evaluation. Nevertheless, extended notions of conversion are desirable, leading to proposals like CoqMT [Strub 2010], where equivalence up to first-order theories is supported. In both cases, the conversion rule is fixed, and extending it requires significant amounts of work. It is thus not possible for users to extend it using their own, domain-specific tactics, and proof objects are thus bound to get large. This is why we have to resort to writing proof scripts.

Checking proof scripts. As mentioned earlier, in order to validate a proof script we need to evaluate it (see Fig. 1a); this is the modus operandi in proof assistants of the HOL family [Harrison 1996; Slind and Norrish 2008]. Therefore, it is easy to extend the checking procedure for proof scripts by writing a new tactic, and calling it as part of a script. The price that this comes to is that there is no way to have any sort of static guarantee about the validity of the script, as proof scripts are completely untyped. This can be somewhat mitigated in Coq by utilizing the static checking that it already supports: the proof checker, and especially, the conversion rule it contains (see Fig. 1b). We can employ proof objects in our scripts; this is especially useful when the proof objects are trivial to write but trigger complex conversion checks. This is the essential idea behind techniques like proof-by-reflection [Boutin 1997], which lead to more robust proof scripts.

In previous work [Stampoulis and Shao 2010] we introduced VeriML, a language that enables programming tactics and proof scripts in a typeful manner using a general-purpose, side-effectful programming model. Combining typed tactics leads to *typed proof scripts*. These are still programs producing proof objects, but the proposition they prove is carried within their type. Information about the current proof state (the set of hypotheses and goals) is also available statically at every intermediate point of the proof script. In this way, the static assurances about proof scripts are significantly increased and many potential sources of type errors are removed. On the other hand, the proof objects contained within the scripts are still checked using a fixed proof checker; this ultimately means that the set of possible static guarantees is still fixed.

Extensible conversion rule. In this paper, we build on our earlier work on VeriML. In order to further increase the amount of static checking of proof scripts that is possible within this language, we propose the notion of an extensible conversion rule (see Fig. 1c). It enables users to write their own domain-specific conversion checks that get included in the conversion rule. This leads to simpler proof scripts, as more parts of the proof can be inferred by the conversion rule and can therefore be omitted. Also, it leads to increased static guarantees for proof scripts, since the conversion checks happen before the rest of the proof script is evaluated.

The way we achieve this is by programming the conversion checks as type-safe tactics within VeriML, and then evaluating them statically using a simple staging mechanism (see Fig. 2). The type of the conversion tactics requires that they produce a proof object which proves the claimed equivalence of the propositions. In this way, type safety of VeriML guarantees that soundness is maintained. At the same time, users are free to extend the conversion rule with their own conversion tactics written in a familiar programming model, without requiring any metatheoretic additions or termination proofs. Such proofs are only necessary if decidability of the extra conversion checks is desired. Furthermore, this approach allows for metatheoretic reductions as the original conversion rule can be programmed within the language. Thus it can be removed from the logic, and replaced by the simpler notion of explicit equalities, leading to both simpler metatheory and a smaller trusted base.

Checking tactics. The above approach addresses the issue of being able to extend the amount of static checking possible for proof scripts. But what about tactics? Our existing work on VeriML shows how the increased type information addresses some of the issues of tactic development using current proof assistants, where tactics are programmed in a completely untyped manner.

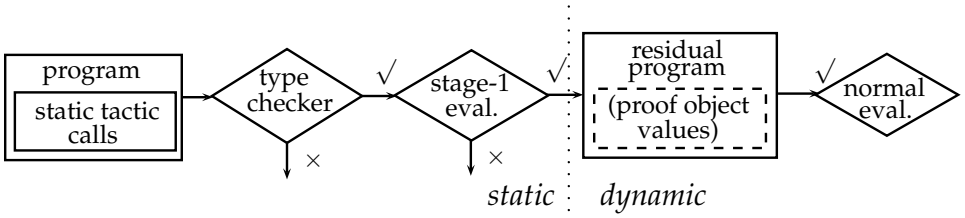


Figure 2. Staging in VeriML

Still, if we consider the case of tactics more closely, we will see that there is a limitation to the amount of checking that is done statically, even using this language. When programming a new tactic, we would like to reuse existing tactics to produce the required proofs. Therefore, rather than writing proof objects by hand inside the code of a tactic, we would rather use proof scripts. The issue is that in order to check whether the contained proof scripts are valid, they need to be evaluated – but this only happens when an invocation of the tactic reaches the point where the proof script is used. Therefore, the static guarantees that this approach provides are severely limited by the fact that the proof scripts inside the tactics cannot be checked statically, when the tactic is defined.

Static proof scripts. This is the second fundamental issue we address in this paper. We show that the same staging construct utilized for introducing the extensible conversion rule, can be leveraged to perform *static proof checking for tactics*. The crucial point of our approach is the proof of existence of a transformation between proof objects, which suggests that under reasonable conditions, a proof script contained within a tactic can be transformed into a static proof script. This static script can then be evaluated at tactic definition time, to be checked for validity.

Last, we will show that this approach lends itself well to writing extensions of the conversion rule. We show that we can create a layering of conversion rules: using a basic conversion rule as a starting point, we can utilize it inside static proof scripts to implicitly prove the required obligations of a more advanced version, and so on. This minimizes the required user effort for writing new conversion rules, and enables truly modular proof checking.

3. Our toolbox

In this section, we will present the essential ingredients that are needed for the rest of our development. The main requirement is a language that supports type-safe manipulation of terms of a particular logic, as well as a general-purpose programming model that includes general recursion and other side-effectful operations. Two recently proposed languages for manipulating LF terms, Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008], fit this requirement, as does VeriML [Stampoulis and Shao 2010], which is a language used to write type-safe tactics. Our discussion is focused on the latter, as it supports a richer ML-style calculus compared to the others, something useful for our purposes. Still, our results apply to all three.

We will now briefly describe the constructs that these languages support, as well as some new extensions that we propose. The interested reader can read more about these constructs in Sec. 6 and in the appendix.

A formal logic. The computational language we are presenting is centered around manipulation of terms of a specific formal logic. We will see more details about this logic in Sec. 4. For the time being, it will suffice to present a set of assumptions about the syntactic classes and typing judgements of this logic, shown in Fig. 3. Logical terms are represented by the syntactic class t, and include proof objects, propositions, terms corresponding to the domain of discourse (e.g. natural numbers), and the needed sorts and type constructors to classify such terms. Their variables are assigned types through an ordered context Φ. A package of a logical term t together with the variables context it inhabits Φ is called a contextual term and denoted as $T = [\Phi]\, t$. Our computational language works over contextual terms for reasons that will be evident later. The logic incorporates

$t ::= \textit{proof object constructors} \mid \textit{propositions} \mid \textit{natural numbers, lists, etc.} \mid \textit{sorts and types} \mid X/\sigma$
$\Phi ::= \bullet \mid \Phi,\ x : t \qquad\qquad T ::= [\Phi]t$
$\Psi ::= \bullet \mid \Psi,\ X : T \qquad\qquad \sigma ::= \bullet \mid \sigma,\ x \mapsto t$
main judgement: $\Psi;\ \Phi \vdash t : t'$ *(type of a logical term)*

Figure 3. Assumptions about the logic language

$k ::= * \mid k_1 \to k_2$
$\tau ::= \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mu\alpha : k.\tau \mid \forall\alpha : k.\tau \mid \alpha \mid \mathsf{array}\ \tau \mid \lambda\alpha : k.\tau \mid \tau_1\ \tau_2 \mid \cdots$
$e ::= () \mid n \mid e_1 + e_2 \mid e_1 \le e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else} e_2 \mid \lambda\mathbf{x} : \tau.e \mid e_1\ e_2 \mid (e_1,\ e_2) \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e$
$\quad \mid \mathsf{case}(e, \mathbf{x}_1.e_1, \mathbf{x}_2.e_2) \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \Lambda\alpha : k.e \mid e\ \tau \mid \mathsf{fix}\ \mathbf{x} : \tau.e \mid \mathsf{mkarray}(e, e') \mid e[e'] \mid e[e'] := e''$
$\quad \mid l \mid \mathsf{error} \mid \cdots$
$\Gamma ::= \bullet \mid \Gamma,\ \mathbf{x} : \tau \mid \Gamma,\ \alpha : k \qquad\qquad \Sigma ::= \bullet \mid \Sigma,\ l : \mathsf{array}\ \tau$

Figure 4. Syntax for the computational language (ML fragment)

$\tau ::= \cdots \mid (X : T) \to \tau \mid (X : T) \times \tau \mid (\phi : \mathsf{ctx}) \to \tau$
$e ::= \cdots \mid \lambda X : T.e \mid e\ T \mid \lambda\phi : \mathsf{ctx}.e \mid e\ \Phi \mid \langle T,\ e\rangle \mid \mathsf{let}\ \langle X,\ \mathbf{x}\rangle = e\ \mathsf{in}\ e'$
$\quad \mid \mathsf{holcase}\ T\ \mathsf{return}\ \tau\ \mathsf{of}\ (T_1 \mapsto e_1) \cdots (T_n \mapsto e_n) \mid \mathsf{ctxcase}\ \Phi\ \mathsf{return}\ \tau\ \mathsf{of}\ (\Phi_1 \mapsto e_1) \cdots (\Phi_n \mapsto e_n)$

Figure 5. Syntax for the computational language (logical term constructs)

such terms by allowing them to get substituted for *meta-variables* X, using the constructor X/σ. When a term $T = [\Phi']t$ gets substituted for X, we go from the Φ' context to the current context Φ using the substitution σ.

Logical terms are classified using other logical terms, based on the normal variables environment Φ, and also an environment Ψ that types meta-variables, thus leading to the $\Psi;\ \Phi \vdash t : t'$ judgement. For example, a term t representing a closed proposition will be typed as $\bullet;\ \bullet \vdash t : \mathsf{Prop}$, while a proof object t_{pf} proving that proposition will satisfy the judgement $\bullet;\ \bullet \vdash t_{\text{pf}} : t$.

ML-style functional programming. We move on to the computational language. As its main core, we assume an ML-style functional language, supporting general recursion, algebraic data types and mutable references (see Fig. 4). Terms of this fragment are typed under a computational variables environment Γ and a store typing environment Σ, mapping mutable locations to types. Typing judgements are entirely standard, leading to a $\Sigma;\ \Gamma \vdash e : \tau$ judgement for typing expressions.

Dependently-typed programming over logical terms. As shown in Fig. 5, the first important additions to the ML computational core are constructs for dependent functions and products over contextual terms T. Abstraction over contextual terms is denoted as $\lambda X : T.e$. It has the dependent function type $(X : T) \to \tau$. The type is dependent since the introduced logical term might be used as the type of another term. An example would be a function that receives a proposition plus a proof object for that proposition, with type: $(P : \mathsf{Prop}) \to (X : P) \to \tau$. Dependent products that package a contextual logical term with an expression are introduced through the $\langle T,\ e\rangle$ construct and eliminated using $\mathsf{let}\ \langle X,\ \mathbf{x}\rangle = e\ \mathsf{in}\ e'$; their type is denoted as $(X : T) \times \tau$. Especially for packages of proof objects with the unit type, we introduce the syntax $\mathsf{LT}(T)$.

Last, in order to be able to support functions that work over terms in any context, we introduce context polymorphism, through a similarly dependent function type over contexts. With these in mind, we can define a simple tactic that gets a packaged proof of a universally quantified formula, and an instantiation term, and returns a proof of the instantiated formula as follows:

$$\begin{array}{l}\mathsf{instantiate}: \ (\phi : \mathsf{ctx},\ T : [\phi]\,\mathsf{Type},\ P : [\phi,\ x : T]\,\mathsf{Prop},\ a : [\phi]\,T) \rightarrow \\ \qquad\qquad \mathsf{LT}([\phi]\,\forall x : T, P) \rightarrow \mathsf{LT}([\phi]\,P/[\mathsf{id}_\phi,\ a])\\ \mathsf{instantiate}\ \phi\ T\ P\ a\ \mathsf{pf}\ = \mathsf{let}\ \langle H \rangle\ = \mathsf{pf}\ \mathsf{in}\ \langle H\ a \rangle\end{array}$$

From here on, we will omit details about contexts and substitutions in the interest of presentation.

Pattern matching over terms. The most important new construct that VeriML supports is a pattern matching construct over logical terms denoted as holcase. This construct is used for dependent matching of a logical term against a set of patterns. The return clause specifies its return type; we omit it when it is easy to infer. Patterns are normal terms that include unification variables, which can be present under binders. This is the essential reason why contextual terms are needed.

Pattern matching over environments. For the purposes of our development, it is very useful to support one more pattern matching construct: matching over logical variable contexts. When trying to construct a certain proof, the logical environment represents what the current proof context is: what the current logical hypotheses at hand are, what types of terms have been quantified over, etc. By being able to pattern match over the environment, we can "look up" things in our current set of hypotheses, in order to prove further propositions. We can thus view the current environment as representing a simple form of the current *proof state*; the pattern matching construct enables us to manipulate it in a type-safe manner.

One example is an "assumption" tactic, that tries to prove a proposition by searching for a matching hypotheses in the context:

$$\begin{array}{l}\mathsf{assumption}: \ (\phi : \mathsf{ctx}, P : \mathsf{Prop}) \rightarrow \mathsf{option}\ \mathsf{LT}(P)\\ \mathsf{assumption}\ \phi\ P = \\ \quad \mathsf{ctxcase}\ \phi\ \mathsf{of}\\ \quad\quad \phi',\ H : P \mapsto \mathsf{return}\ \langle H \rangle\\ \quad\quad \mid \phi',\ \_ \quad\ \mapsto \mathsf{assumption}\ \phi'\ P\end{array}$$

Proof object erasure semantics (new feature). The only construct that can influence the evaluation of a program based on the structure of a logical term is the pattern matching construct. For our purposes, pattern matching on proof objects is not necessary – we never look into the structure of a completed proof. Thus we can have the typing rules of the pattern matching construct specifically disallow matching on proof objects.

In that case, we can define an alternate operational semantics for our language where all proof objects are *erased* before using the original small-step reduction rules. Because of type safety, these proof-erasure semantics are guaranteed to yield equivalent results: even if no proof objects are generated, they are still bound to exist.

Implicit arguments. Let us consider again the *instantiate* function defined earlier. This function expects five arguments. From its type alone, it is evident that only the last two arguments are strictly necessary. The last argument, corresponding to a proof expression for the proposition $\forall x : T, P$, can be used to reconstruct exactly the arguments ϕ, T and P. Furthermore, if we know what the resulting type of a call to the function needs to be, we can choose even the instantiation argument a appropriately. We employ a simple inferrence mechanism so that such arguments are omitted from our programs. This feature is also crucial in our development in order to implicitly maintain and utilize the current proof state within our proof scripts.

$$
\begin{array}{rrl}
\text{(sorts)} & s & ::= \mathsf{Type} \mid \mathsf{Type}' \\
\text{(kinds)} & \mathcal{K} & ::= \mathsf{Prop} \mid \mathsf{Nat} \mid \mathcal{K}_1 \to \mathcal{K}_2 \\
\text{(props.)} & P & ::= P_1 \to P_2 \mid \forall x : \mathcal{K}.P \mid x \mid \mathsf{True} \mid \mathsf{False} \mid P_1 \wedge P_2 \mid \cdots \\
\text{(dom.obj.)} & d & ::= \mathsf{Zero} \mid \mathsf{Succ}\ d \mid P \mid \cdots \\
\text{(proof objects)} & \pi & ::= x \mid \lambda x : P.\pi \mid \pi_1\ \pi_2 \mid \lambda x : \mathcal{K}.\pi \mid \pi\ d \mid \cdots \\
\text{(HOL terms)} & t & ::= s \mid \mathcal{K} \mid P \mid d \mid \pi
\end{array}
$$

Selected rules:

$$
\frac{\Psi;\ \Phi, x : P \vdash \pi : P'}{\Psi;\ \Phi \vdash \lambda x : P.\pi : P \to P'} \to \textsc{Intro}
\qquad
\frac{\Psi;\ \Phi \vdash \pi : P \to P' \qquad \Psi;\ \Phi \vdash \pi' : P}{\Psi;\ \Phi \vdash \pi\ \pi' : P'} \to \textsc{Elim}
$$

Figure 6. Syntax and selected rules of the logic language λHOL

$$
\textsc{Conversion}\ \frac{\Psi;\ \Phi \vdash_c \pi : P \qquad P =_{\beta\mathbb{N}} P'}{\Psi;\ \Phi \vdash_c \pi : P'}
$$

$d \to_{\beta\mathbb{N}} d'$:

$$
\begin{array}{l}
(\lambda x : \mathcal{K}.d)\ d' \to_{\beta\mathbb{N}} d[d'/x] \\
\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ \mathsf{zero} \to_{\beta\mathbb{N}} d_z \\
\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ (\mathsf{succ}\ d) \to_{\beta\mathbb{N}} d_s\ d\ (\mathsf{natElim}_{\mathcal{K}}\ d_z\ d_s\ d)
\end{array}
$$

$d =_{\beta\mathbb{N}} d'$: is the compatible, reflexive, symmetric and transitive closure of $d \to_{\beta\mathbb{N}} d'$

Figure 7. Extending λHOL with the conversion rule (λHOL_c)

Minimal staging support (new feature). Using the language we have seen so far we are able to write powerful tactics using a general-purpose programming model. But what if, inside our programs, we have calls to tactics where all of their arguments are constant? Presumably, those tactic calls could be evaluated to proof objects prior to tactic invocation. We could think of this as a form of generalized constant folding, which has one intriguing benefit: we can tell statically whether the tactic calls succeed or not.

This paper is exactly about exploring this possibility. Towards this effect, we introduce a rudimentary staging construct in our computational language. This takes the form of a letstatic construct, which binds a static expression to a variable. The static expression is evaluated during stage one (see Fig. 2), and can only depend on other static expressions. Details of this construct are presented in Fig. 11d and also in Sec. 6. After this addition, expressions in our language have a three-phase lifetime, that are also shown in Fig. 2.

- type-checking, where the well-formedness of expressions according to the rules of the language is checked, and inference of implicit arguments is performed
- static evaluation, where expressions inside letstatic are reduced to values, yielding a residual expression
- run-time, where the residual expression is evaluated

4. Extensible conversion rule

With these tools at hand, let us now return to the first issue that motivates us: the fact that proof checking is rigid and cannot be extended with user-defined procedures. As we have said in our introduction, many modern proof assistants are based on logics that include a *conversion rule*. This rule essentially identifies propositions up to some equivalence relation: usually this is equivalence up to partial evaluation of the functions contained within propositions.

The supported relation is decided when the logic is designed. Any extension to this relation requires a significant amount of work, both in terms of implementation, and in terms of metatheoretic proof required. This is evidenced by projects that extend the conversion rule in Coq, such as Blanqui et al. [1999] and Strub [2010]. Even if user extensions are supported, those only take the form of first-order theories. Can we do better than this, enabling arbitrarily complex user extensions, written with the full power of ML, yet maintaining soundness?

It turns out that we can: this is the subject of this section. The key idea is to recognize that the conversion rule is essentially a tactic, embedded within the type checker of the logic. Calls to this tactic are made implicitly as part of checking a given proof object for validity. So how can we support a flexible, extensible alternative? Instead of hardcoding a conversion tactic within the logic type checker, we can program a type-safe version of the same tactic within VeriML, with the requirement that it provides proof of the claimed equivalence. Instead of calling the conversion tactic as part of proof checking, we use staging to call the tactic statically – after (VeriML) type checking, but before runtime execution. This can be viewed as a second, potentially non-terminating proof checking stage. Users are now free to write their own conversion tactics, extending the static checking available for proof objects and proof scripts. Still, soundness is maintained, since full proof objects in the original logic can always be constructed. As an example, we have extended the conversion rule that we use by a congruence closure procedure, which makes use of mutable data structures, and by an arithmetic simplification procedure.

4.1 Introducing: the conversion rule

First, let us present what the conversion rule really is in more detail. We will base our discussion on a simple type-theoretic higher-order logic, based on the λHOL logic as described in Barendregt and Geuvers [1999], and used in our original work on VeriML [Stampoulis and Shao 2010]. We can think of such a logic composed by the following broad classes: the objects of the domain of discourse d, which are the objects that the logic reasons about, such as natural numbers and lists; their classifiers, the kinds $\mathcal{K}$ (classified in turn by sorts s); the propositions P; and the derivations, which prove that a certain proposition is true. We can represent derivations in a linear form as terms π in a typed lambda-calculus; we call such terms proof objects, and their types represent propositions in the logic. Checking whether a derivation is a valid proof of a certain proposition amounts to type-checking its corresponding proof object. Some details of this logic are presented in Fig. 6; the interested reader can find more information about it in the above references and in the appendix (Sec. A).

In Fig. 6, we show what the conversion rule looks like for this logic: it is a typing judgement that effectively identifies propositions up to an equivalence relation, with respect to checking proof objects. We call this version of the logic λHOL_c and use $\vdash_c$ to denote its entailment relation. The equivalence relation we consider in the conversion rule is evaluation up to β-reductions and uses of primitive recursion of natural numbers, denoted as $\mathsf{natElim}$. In this way, trivial arguments based on this notion of computation alone need not be witnessed, as for example is the fact that $(\mathsf{Succ}\ x) + y = \mathsf{Succ}\ (x + y)$ – when the addition function is defined by primitive recursion on the first argument. Of course, this is only a very basic use of the conversion rule. It is possible to omit larger proofs through much more sophisticated uses. This leads to simpler proofs and smaller proof objects.

Still, when using this approach, the choice of what relation is supported by the conversion rule needs to be made during the definition of the logic. This choice permeates all aspects of the metatheory of the logic. It is easy to see why, even with the tiny fragment of logic we have introduced. Most typing rules for proof objects in

$$\frac{\Psi;\ \Phi \vdash_e d_1 : \mathcal{K} \qquad \Psi;\ \Phi \vdash_c d_2 : \mathcal{K}}{\Psi;\ \Phi \vdash_e d_1 = d_2 : \mathsf{Prop}} \qquad \frac{\Psi;\ \Phi \vdash_e d : \mathcal{K}}{\Psi;\ \Phi \vdash_e \mathsf{refl}\ d : d = d}$$

$$\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e P : \mathsf{Prop} \qquad \Psi;\ \Phi \vdash_e d_1 : \mathcal{K} \qquad \Psi;\ \Phi \vdash_e \pi : P[d_1/x] \qquad \Psi;\ \Phi \vdash_e \pi' : d_1 = d_2}{\Psi;\ \Phi \vdash_e \mathsf{leibniz}\ (\lambda x : \mathcal{K}.P)\ \pi\ \pi' : P[d_2/x]}$$

$$\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e \pi : d_1 = d_2}{\Psi;\ \Phi \vdash_e \mathsf{lamEq}\ (\lambda x : \mathcal{K}.\pi) : (\lambda x : \mathcal{K}.d_1) = (\lambda x : \mathcal{K}.d_2)}$$

$$\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e \pi : d_1 = d_2 \qquad \Psi;\ \Phi \vdash_e d_1 : \mathsf{Prop}}{\Psi;\ \Phi \vdash_e \mathsf{forallEq}\ (\lambda x : \mathcal{K}.\pi) : (\forall x : \mathcal{K}.d_1) = (\forall x : \mathcal{K}.d_2)}$$

$$\frac{\Psi;\ \Phi,\ x : \mathcal{K} \vdash_e d : \mathcal{K}' \qquad \Psi;\ \Phi \vdash_e d' : \mathcal{K}}{\Psi;\ \Phi \vdash_e \mathsf{betaEq}\ (\lambda x : \mathcal{K}.d)\ d' : (\lambda x : \mathcal{K}.d)\ d' = d[d'/x]}$$

Axioms assumed:

$$\begin{array}{lcl} \mathsf{natElimBase}_{\mathcal{K}} & : & \forall f_z.\forall f_s.\mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ \mathsf{zero} = f_z \\ \mathsf{natElimStep}_{\mathcal{K}} & : & \forall f_z.\forall f_s.\forall n.\ \mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ (\mathsf{succ}\ n) = \\ & & \qquad f_s\ n\ (\mathsf{natElim}_{\mathcal{K}}\ f_z\ f_s\ n) \end{array}$$

Figure 8. Extending λHOL with explicit equality ($\lambda\mathrm{HOL}_e$)

the logic are similar to the rules →INTRO and →ELIM: they are syntax-directed. This means that upon seeing the associated proof object constructor, like $\lambda x : P.\pi$ in the case of →INTRO, we can directly tell that it applies. If all rules were syntax directed, it would be entirely simple to prove that the logic is sound by an inductive argument: essentially, since no proof constructor for False exists, there is no valid derivation for False.

In this logic, the only rule that is not syntax directed is exactly the conversion rule. Therefore, in order to prove the soundness of the logic, we have to show that the conversion rule does not somehow introduce a proof of False. This means that proving the soundness of the logic passes essentially through the specific relation we have chosen for the conversion rule. Therefore, this approach is foundationally limited from supporting user extensions, since any new extension would require a new metatheoretic result in order to make sure that it does not violate logical soundness.

4.2 Throwing conversion away

Since having a fixed conversion rule is bound to fail if we want it to be extensible, what choice are we left with, but to throw it away? This radical sounding approach is what we will do here. We can replace the conversion rule by an explicit notion of equality, and provide explicit proof witnesses for rewriting based on that equality. Essentially, all the points where the conversion rule was alluded to and proofs were omitted, need now be replaced by proof objects witnessing the equivalence. Some details for the additions required to the base λHOL logic are shown in Fig. 8, yielding the $\lambda\mathrm{HOL}_e$ logic. There are good reasons for choosing this version: first, the proof checker is as simple as possible, and does not need to include the conversion checking routine. We could view this routine as performing proof search over the replacement rules, so it necessarily is more complicated, especially since it needs to be relatively efficient. Also, the metatheory of the logic itself can be simplified. Even when the conversion rule is supported, the metatheory for the associated logic is proved through the explicit

```
βℕequal : (ϕ : ctx, T : Type, t1 : T, t2 : T) → option LT(t1 = t2)
βℕequal ϕ T t1 t2 =
  holcase whnf ϕ T t1, whnf ϕ T t2 of
    ((ta : T' → T) tb), (tc td) ↦
      do ⟨pf1⟩ ← βℕequal ϕ (T' → T) ta tc
         ⟨pf1⟩ ← βℕequal ϕ T' tb td
         return ⟨··· proof of ta tb = tc td ···⟩
  | (ta → tb), (tc → td) ↦
      do ⟨pf1⟩ ← βℕequal ϕ Prop ta tc
         ⟨pf1⟩ ← βℕequal ϕ Prop tb td
         return ⟨··· proof of ta → tb = tc → td ···⟩
  | (λx : T.t1), (λx : T.t2) ↦
      do ⟨pf⟩ ← βℕequal [ϕ, x : T] Prop t1 t2
         return ⟨··· proof of λx : T.t1 = λx : T.t2 ···⟩
  | t1, t1 ↦ do return ⟨··· proof of t1 = t1 ···⟩
  | t1, t2 ↦ None

requireEqual : (ϕ : ctx, T : Type, t1 : T, t2 : T).LT(t1 = t2)
requireEqual ϕ T t1 t2 =
  match βℕequal ϕ T t1 t2 with Some x ↦ x | None ↦ error
```

Figure 9. VeriML tactic for checking equality up to β-conversion

equality approach; this is because model construction for a logic benefits from using explicit equality [Siles and Herbelin 2010].

Still, this approach has a big disadvantage: the proof objects soon become extremely large, since they include painstakingly detailed proofs for even the simplest of equivalences. This precludes their use as independently checkable proof certificates that can be sent to a third party. It is possible that this is one of the reasons why systems based on logics with explicit equalities, such as HOL4 [Slind and Norrish 2008] and Isabelle/HOL [Nipkow et al. 2002], do not generate proof objects by default.

4.3 Getting conversion back

We will now see how it is possible to reconcile the explicit equality based approach with the conversion rule: we will gain the conversion rule back, albeit it will remain completely outside the logic. Therefore we will be free to extend it, all the while without risking introducing unsoundness in the logic, since the logic remains fixed (λHOL_e as presented above).

We do this by revisiting the view of the conversion rule as a special "trusted" tactic, through the tools presented in the previous section. First, instead of hardcoding a conversion tactic in the type checker, we program a *type-safe conversion tactic*, utilizing the features of VeriML. Based on typing alone we require that it returns a valid proof of the claimed equivalences:

$$\beta\mathbb{N}\text{equal} : (\phi : \text{ctx},\ T : \text{Type},\ t : T,\ t' : T) \rightarrow \text{option}\ \text{LT}(t = t')$$

Second, we evaluate this tactic under *proof erasure semantics*. This means that no proof objects are produced, leading to the same space gains as the original conversion rule. Third, we use the staging construct in order to *check conversion statically*.

```
whnf : (ϕ : ctx, T : Type, t : T) → (t' : T) × LT(t = t')
whnf ϕ T t = holcase t of
  (t1 : T' → T)(t2 : T') ↦
    let ⟨t1', pf1⟩ = whnf ϕ (T' → T) t1 in
    holcase t1' of
      λx : T'.tf  ↦  ⟨[ϕ] tf/[id_Φ, t2], ···⟩
      | t1'       ↦  ⟨[ϕ] t1' t2, ···⟩
  | natElim_K fz fs n ↦
      let ⟨n', pf1⟩ = whnf ϕ Nat n in holcase n' of
            zero      ↦  ⟨[ϕ] fz, ···⟩
          | succ n'   ↦  ⟨[ϕ] fs n' (natElim_K fz fs n'), ···⟩
          | n'        ↦  ⟨[ϕ] natElim_K fz fs n', ···⟩
  | t ↦ ⟨t, ···⟩
```

Figure 10. VeriML tactic for rewriting to weak head-normal form

Details. We now present our approach in more detail. First, in Fig. 9, we show a sketch of the code behind the type-safe conversion check tactic. It works by first rewriting its input terms into weak head-normal form, via the whnf function in Fig. 10, and then recursively checking their subterms for equality. In the equivalence checking function, more cases are needed to deal with quantification; while in the rewriting procedure, a recursive call is missing, which would complicate our presentation here. We also define a version of the tactic that raises an error instead of returning an option type if we fail to prove the terms equal, which we call requireEqual. The full details can be found in our implementation.

The code of the $\beta\mathbb{N}$equal tactic is in fact entirely similar to the code one would write for the conversion check routine inside a logic type checker, save for the extra types and proof objects. It therefore follows trivially that everything that holds for the standard implementation of the conversion check also holds for this code: e.g. it corresponds exactly to the $=_{\beta\mathbb{N}}$ relation as defined in the logic; it is bound to terminate because of the strong normalization theorem for this relation; and its proof-erased version is at least as trustworthy as the standard implementation.

Furthermore, given this code, we can produce a form of *typed proof scripts* inside VeriML that correspond exactly to proof objects in the logic with the conversion rule, both in terms of their actual code, and in terms of the steps required to validate them. This is done by constructing a proof script in VeriML by induction on the derivation of the proof object in λHOL_c, replacing each proof object constructor by an equivalent VeriML tactic as follows:

constructor	to tactic	of type
$\lambda x : P.\pi$	Assume e	$\mathsf{LT}([\phi,\ H : P]\,P') \to \mathsf{LT}(P \to P')$
$\pi_1\ \pi_2$	Apply $e_1\ e_2$	$\mathsf{LT}(P \to P') \to \mathsf{LT}(P) \to \mathsf{LT}(P')$
$\lambda x : \mathcal{K}.\pi$	Intro e	$\mathsf{LT}([\phi,\ x : T]\,P') \to \mathsf{LT}(\forall x : T, P')$
$\pi\ d$	Inst $e\ a$	$\mathsf{LT}(\forall x : T, P) \to (a : T) \to \mathsf{LT}(P/[\mathsf{id},\ a])$
c	Lift c	$(H : P) \to \mathsf{LT}(P)$
(conversion)	Conversion	$\mathsf{LT}(P) \to \mathsf{LT}(P = P') \to \mathsf{LT}(P')$

Here we have omitted the current logical environment ϕ; it is maintained through syntactic means as discussed in Sec. 7 and through type inference. The only subtle case is conversion. Given the transformed proof e for the

proof object π contained within a use of the conversion rule, we call the conversion tactic as follows:

$$\mathsf{letstatic\ pf\ =\ requireEqual}\ P\ P'\ \mathsf{in\ Conversion\ e\ pf}$$

The arguments to requireEqual can be easily inferred, making crucial use of the rich type information available. Conversion could also be used implicitly in the other tactics. Thus the resulting expression looks entirely identical to the original proof object.

Correspondence with original proof object. In order to elucidate the correspondence between the resulting proof script expression and the original proof object, it is fruitful to view the proof script as a proof certificate, sent to a third party. The steps required to check whether it constitutes a valid proof are the following. First, the whole expression is checked using the type checker of the computational language. Then, the calls to the requireEqual function are evaluated during stage one, using proof erasure semantics. We expect them to be successful, just as we would expect the conversion rule to be applicable when it is used. Last, the rest of the tactics are evaluated; by a simple argument, based on the fact that they do not use pattern matching or side-effects, they are guaranteed to terminate and produce a proof object in $\lambda\mathrm{HOL}_e$. This validity check is entirely equivalent to the behavior of type-checking the $\lambda\mathrm{HOL}_c$ proof object, save for pushing all conversion checks towards the end.

4.4 Extending conversion at will

In our treatment of the conversion rule we have so far focused on regaining the $\beta\mathbb{N}$ conversion in our framework. Still, there is nothing confining us to supporting this conversion check only. As long as we can program a conversion tactic in VeriML that has the right type, it can safely be made part of our conversion rule.

For example, we have written an eufEqual function, which checks terms for equivalence based on the equality with uninterpreted functions decision procedure. It is adapted from our previous work on VeriML [Stampoulis and Shao 2010]. This equivalence checking tactic isolates hypotheses of the form $d_1 = d_2$ from the current context, using the newly-introduced context matching support. Then, it constructs a union-find data structure in order to form equivalence classes of terms. Based on this structure, and using code similar to $\beta\mathbb{N}$equal (recursive calls on subterms), we can decide whether two terms are equal up to simple uses of the equality hypotheses at hand. We have combined this tactic with the original $\beta\mathbb{N}$equal tactic, making the implicit equivalence supported similar to the one in the Calculus of Congruent Constructions [Blanqui et al. 2005]. This demonstrates the flexibility of this approach: equivalence checking is extended with a sophisticated decision procedure, which is programmed using its original, imperative formulation. We have programmed both the rewriting procedure and the equality checking procedure in an extensible manner, so that we can globally register further extensions.

4.5 Typed proof scripts as certificates

Earlier we discussed how we can validate the proof scripts resulting from turning the conversion rule into explicit tactic calls. This discussion shows an interesting aspect of typed proof scripts: they can be viewed as a proof witness that is a flexible compromise between untyped proof scripts and proof objects. When a typed proof script consists only of static calls to conversion tactics and uses of total tactics, it can be thought of as a proof object in a logic with the corresponding conversion rule. When it also contains other tactics, that perform potentially expensive proof search, it corresponds more closely to an untyped proof script, since it needs to be fully evaluated. Still, we are allowed to validate parts of it statically. This is especially useful when developing the proof script, because we can avoid the evaluation of expensive tactic calls while we focus on getting the skeleton of the proof correct.

Using proof erasure for evaluating requireEqual is only one of the choices the receiver of such a proof certificate can make. Another choice would be to have the function return an actual proof object, which we can check using the $\lambda\mathrm{HOL}_e$ type checker. In that case, the VeriML interpreter does not need to become part of

the trusted base of the system. Last, the 'safest possible' choice would be to avoid doing any evaluation of the function, and ask the proof certificate provider to do the evaluation of requireEqual themselves. In that case, no evaluation of computational code would need to happen at the proof certificate receiver's side. This mitigates any concerns one might have for code execution as part of proof validity checking, and guarantees that the small λHOL_e type checker is the trusted base in its entirety. Also, the receiver can decide on the above choices selectively for different conversion tactics – e.g. use proof erasure for $\beta\mathbb{N}$equal but not for eufEqual, leading to a trusted base identical to the λHOL_c case. This means that the choice of the conversion rule rests with the proof certificate receiver and not with the designer of the logic. Thus the proof certificate receiver can choose the level of trust they require at will.

5. Static proof scripts

In the previous section, we have demonstrated how proof checking for typed proof scripts can be made user-extensible, through a new treatment of the conversion rule. It makes use of user-defined, type-safe tactics, which are evaluated statically. The question that remains is what happens with respect to proofs within tactics. If a proof script is found within a tactic, must we wait until that evaluation point is reached to know whether the proof script is correct or not? Or is there a way to check this statically, as soon as the tactic is defined?

In this section we show how this is possible to do in VeriML using the staging construct we have introduced. Still, in this case matters are not as simple as evaluating certain expressions statically rather than dynamically. The reason is that proof scripts contained within tactics mention uninstantiated meta-variables, and thus cannot be evaluated through staging. We resolve this by showing the existence of a transformation, which "collapses" logical terms from an arbitrary meta-variables context into the empty one.

We will focus on the case of developing conversion routines, similar to the ones we saw earlier. The ideas we present are generally applicable when writing other types of tactics as well; we focus on conversion routines in order to demonstrate that the two main ideas we present in this paper can work in tandem.

A rewriter for plus. We will consider the case of writing a rewriter –similar to whnf– for simplifying expressions of the form $x+y$, depending on the second argument. The addition function is defined by induction on the first argument, as follows:

$$(+) = \lambda x.\lambda y.\mathsf{natElim}_{\mathsf{Nat}}\ y\ (\lambda p.\lambda r.\mathsf{Succ}\ r)\ x$$

In order for rewriters to be able to use existing as well as future rewriters to perform their recursive calls, we write them in the open recursion style – they receive a function of the same type that corresponds to the "current" rewriter. The code looks as follows:

$$
\begin{array}{l}
\mathsf{rewriterType} = (\phi : \mathsf{ctx}, T : \mathsf{Type}, t : T) \to (t' : T) \times \mathsf{LT}(t = t') \\
\mathsf{plusRewriter1} : \ \mathsf{rewriterType} \to \mathsf{rewriterType} \\
\mathsf{plusRewriter1}\ \mathsf{recursive}\ \phi\ T\ t = \mathsf{holcase}\ t\ \mathsf{with} \\
\quad x+y \mapsto \\
\qquad \mathsf{let}\ \langle y',\ \langle \mathsf{pfy}' \rangle \rangle = \mathsf{recursive}\ \phi\ y\ \mathsf{in} \\
\qquad \mathsf{let}\ \langle t',\ \langle \mathsf{pft}' \rangle \rangle\ = \\
\qquad\quad \mathsf{holcase}\ y'\ \mathsf{return}\ \Sigma t' : [\phi]\,\mathsf{Nat.LT}([\phi]\,x+y' = t')\ \mathsf{of} \\
\qquad\quad\ \ 0 \qquad \mapsto \langle x, \cdots \textit{ proof of } x+0 = x \cdots \rangle \\
\qquad\quad \mid \mathsf{Succ}\ y' \mapsto \Big\langle \mathsf{Succ}(x+y'), \\
\qquad\quad\qquad\qquad \cdots \textit{ proof of } x+\mathsf{Succ}\ y' = \mathsf{Succ}\ (x+y') \cdots \Big\rangle \\
\qquad\quad\ \ \mid y' \qquad \mapsto \langle x+y', \cdots \textit{ proof of } x+y' = x+y' \cdots \rangle \\
\qquad \mathsf{in}\ \langle t',\ \langle \cdots \textit{ proof of } x+y = t' \cdots \rangle \rangle \\
\quad \mid t \qquad \mapsto \langle t, \cdots \textit{ proof of } t = t \cdots \rangle
\end{array}
$$

While developing such a tactic, we can leverage the VeriML type checker to know the types of missing proofs. But how do we fill them in? For the interesting cases of $x+0=x$ and $x+\mathsf{Succ}\ y' = \mathsf{Succ}\ (x+y')$, we would certainly need to prove the corresponding lemmas. But for the rest of the cases, the corresponding lemmas would be uninteresting and tedious to state, such as the following for the $x+y=t'$ case:

$$\mathsf{lemma1} : \forall x,y,y',t', y=y' \rightarrow (x+y'=t') \rightarrow x+y=t$$

Stating and proving such lemmas soon becomes a hindrance when writing tactics. An alternative is to use the congruence closure conversion rule to solve this trivial obligation for us directly at the point where it is required. Our first attempt would be:

$$\begin{array}{l}\text{proof of } x+y=t' \ \equiv \\ \quad \mathsf{let}\ \langle \mathsf{pf} \rangle = \mathsf{requireEqual}\ \ [\phi, H_1 : y=y', H_2 : x+y'=t']\ \ (x+y)\ \ t' \\ \quad \mathsf{in}\ \left\langle [\phi]\,\mathsf{pf}/[\mathsf{id}_\phi,\ \mathsf{pfy}',\ \mathsf{pft}'] \right\rangle\end{array}$$

The benefit of this approach is evident when utilizing implicit arguments, since most of the details can be inferred and therefore omitted. Here we had to alter the environment passed to requireEqual, which includes several extra hypotheses. Once the resulting proof has been computed, the hypotheses are substituted by the actual proofs that we have.

The problem with this approach is two-fold: first, the call to the requireEqual tactic is recomputed every time we reach that point of our function. For such a simple tactic call, this does not impact the runtime significantly; still, if we could avoid it, we would be able use more sophisticated and expensive tactics. The second problem is that if for some reason the requireEqual is not able to prove what it is supposed to, we will not know until we actually reach that point in the function.

Moving to static proofs. This is where using the letstatic construct becomes essential. We can evaluate the call to requireEqual statically, during stage one interpretation. Thus we will know at the time that plusRewriter1 is defined whether the call succeeded; also, it will be replaced by a concrete value, so it will not affect the runtime behavior of each invocation of plusRewriter1 anymore. To do that, we need to avoid mentioning any of the metavariables that are bound during runtime, like x, y, and t'. This is done by specifying an appropriate environment in the call to requireEqual, similarly to the way we incorporated the extra knowledge above and substituted it later. Using this approach, we have:

$$\begin{array}{l}\text{proof of } x+y=t' \ \equiv \\ \quad \mathsf{letstatic}\ \langle \mathsf{pf} \rangle\ = \\ \quad\ \ \mathsf{let}\ \phi'\ = [x,y,y',t' : \mathsf{Nat}, H_1 : y=y', H_2 : x+y'=t']\ \mathsf{in} \\ \quad\ \ \mathsf{requireEqual}\ \phi'\ (x+y)\ t' \\ \quad \mathsf{in}\ \left\langle [\phi]\,\mathsf{pf}/[x/\mathsf{id}_\phi, y/\mathsf{id}_\phi, y'/\mathsf{id}_\phi, t'/\mathsf{id}_\phi, \mathsf{pfy}'/\mathsf{id}_\phi, \mathsf{pft}'/\mathsf{id}_\phi] \right\rangle\end{array}$$

What we are essentially doing here is replacing the meta-variables by normal logical variables, which our tactics can deal with. The meta-variable context is "collapsed" into a normal context; proofs are constructed using tactics in this environment; last, the resulting proofs are transported back into the desired context by substituting meta-variables for variables. We have explicitly stated the substitutions in order to distinguish between normal logical variables and meta-variables.

The reason why this transformation needs to be done is that functions in our computational language can only manipulate logical terms that are open with respect to a normal variables context; not logical terms that are open with respect to the meta-variables context too. A much more complicated, but also more flexible alternative to using this "collapsing" trick would be to support meta-n-variables within our computational language directly.

Overall, this approach is entirely similar to proving the auxiliary lemma mentioned above, prior to the tactic definition. The benefit is that by leveraging the type information together with type inference, we can avoid

stating such lemmas explicitly, while retaining the same runtime behavior. We thus end up with very concise proof expressions that are statically validated. We introduce syntactic sugar for binding a static proof script to a variable, and then performing a substitution to bring it into the current context, since this is a common operation.

$$\langle e \rangle_{\text{static}} \equiv \text{letstatic } \langle \text{pf} \rangle = e \text{ in } \langle [\phi]\, \text{pf}/\cdots \rangle$$

Based on these, the trivial proofs in the above tactic can be filled in using a simple $\langle \text{requireEqual} \rangle_{\text{static}}$ call; for the other two we use $\langle \text{Instantiate } (\text{NatInduction requireEqual requireEqual})\ x \rangle_{\text{static}}$.

After we define plusRewriter1, we can register it with the global equivalence checking procedure. Thus, all later calls to requireEqual will benefit from this simplification. It is then simple to prove commutativity for addition:

$$\begin{array}{lcl} \text{plusComm} & : & \text{LT}(\forall x, y. x + y = y + x) \\ \text{plusComm} & = & \text{NatInduction requireEqual requireEqual} \end{array}$$

Based on this proof, we can write a rewriter that takes commutativity into account and uses the hash values of logical terms to avoid infinite loops. We have worked on an arithmetic simplification rewriter that is built by layering such rewriters together, using previous ones to aid us in constructing the proofs required in later ones. It works by converting expressions into a list of monomials, sorting the list based on the hash values of the variables, and then factoring monomials on the same variable. Also, the eufEqual procedure mentioned earlier has all of its associated proofs automated through static proof scripts, using a naive, potentially non-terminating, equality rewriter.

Is collapsing always possible? A natural question to ask is whether collapsing the metavariables context into a normal context is always possible. In order to cast this as a more formal question, we notice that the essential step is replacing a proof object π of type $[\Phi]\, t$, typed under the meta-variables environment Ψ, by a proof object π' of type $[\Phi']\, t'$ typed under the empty meta-variables environment. There needs to be a substitution so that π' gets transported back to the Φ, Ψ environment, and has the appropriate type.

We have proved that this is possible under certain restrictions: the types of the metavariables in the current context need to depend on the same free variables context Φ_{max}, or prefixes of that context. Also the substitutions they are used with need to be prefixes of the identity substitution for Φ_{max}. Such terms are characterized as collapsible. We have proved that collapsible terms can be replaced using terms that do not make use of metavariables; more details can be found in Sec. 6 and in Sec. F of the appendix.

This restriction corresponds very well to the treatment of variable contexts in the Delphin language. This language assumes an ambient context of logical variables, instead of full, contextual modal terms. Constructs to extend this context and substitute a specific variable exist. If this last feature is not used, the ambient context grows monotonically and the mentioned restriction holds trivially. In our tests, this restriction has not turned out to be limiting.

6. Metatheory

We have completed an extensive reworking of the metatheory of VeriML, in order to incorporate the features that we have presented in this paper. Our new metatheory includes a number of technical advances compared to our earlier work [Stampoulis and Shao 2010]. We will present a technical overview of our metatheory in this section; full details can be found in the appendix.

Variable representation technique. Though our metatheory is done on paper, we have found that using a concrete variable representation technique elucidates some aspects of how different kinds of substitutions work in our language, compared to having normal named variables. For example, instantiating a context variable with

Syntax of the logic

$$\begin{array}{l}\textit{(terms)}\ t ::= s \mid c \mid f_i \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \mid \mathsf{refl}\ t \mid \mathsf{leibniz}\ t_1\ t_2 \mid \mathsf{lamEq}\ t \mid \mathsf{forallEq}\ t_1\ t_2 \mid \mathsf{betaEq}\ t_1\ t_2 \\ \textit{(sorts)}\ s ::= \mathsf{Prop} \mid \mathsf{Type} \mid \mathsf{Type}' \qquad \textit{(var. context)}\ \ \Phi ::= \bullet \mid \Phi,\ t \qquad \textit{(substitutions)}\ \ \sigma ::= \bullet \mid \sigma,\ t\end{array}$$

Example of representation: $a : \mathsf{Nat} \vdash \lambda x : \mathsf{Nat}.(\lambda y : \mathsf{Nat}.\mathsf{refl}\ (\mathsf{plus}\ a\ y))(\mathsf{plus}\ a\ x) \mapsto \mathsf{Nat} \vdash \lambda(\mathsf{Nat}).(\lambda(\mathsf{Nat}).\mathsf{refl}\ (\mathsf{plus}\ f_0\ b_0))\ (\mathsf{plus}\ f_0\ b_0)$

Freshen: $\lceil t \rceil^n_m$

$$\begin{array}{lcl}\lceil f_i \rceil^n_m & = & f_i \\ \lceil b_n \rceil^n_m & = & f_m \\ \lceil b_i \rceil^n_m & = & b_i \text{ when } i < n \\ \lceil (\lambda(t_1).t_2) \rceil^n & = & \lambda(\lceil t_1 \rceil^n).\ \lceil t_2 \rceil^{n+1} \\ \lceil t_1\ t_2 \rceil & = & \lceil t_1 \rceil\ \lceil t_2 \rceil\end{array}$$

Bind: $\lfloor t \rfloor^n_m$

$$\begin{array}{lcl}\lfloor f_{m-1} \rfloor^n_m & = & b_n \\ \lfloor f_i \rfloor^n_m & = & f_i \text{ when } i < m-1 \\ \lfloor b_i \rfloor & = & b_{i+1} \\ \lfloor (\lambda(t_1).t_2) \rfloor & = & \lambda(\lfloor t_1 \rfloor^n).\ \lfloor t_2 \rfloor^{n+1} \\ \lfloor t_1\ t_2 \rfloor & = & \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor\end{array}$$

(a) Hybrid deBruijn levels-deBruijn indices representation technique

---

Syntax

$$\begin{array}{l}t ::= \cdots \mid f_{\mathbf{I}} \mid X_i/\sigma \qquad \Phi ::= \bullet \mid \Phi,\ t \mid \Phi,\ \phi_i \qquad \sigma ::= \bullet \mid \sigma,\ t \mid \sigma,\ \mathbf{id}(\phi_i) \qquad \textit{(indices)}\ \mathbf{I} ::= n \mid \mathbf{I} + |\phi_i| \qquad \textit{(ctx.terms)}\ T ::= [\Phi]t \mid [\Phi]\Phi' \\ \textit{(ctx.kinds)}\ K ::= [\Phi]t \mid [\Phi]\,\mathsf{ctx} \qquad \textit{(extension context)}\ \Psi ::= \bullet \mid \Psi,\ K \qquad \textit{(ext. subst.)}\ \sigma_\Psi ::= \bullet \mid \sigma_\Psi,\ T\end{array}$$

$\Psi;\ \Phi \vdash t : t'$ (sample)

$$\frac{\Phi.\mathbf{I} = t}{\Psi;\ \Phi \vdash f_{\mathbf{I}} : t} \qquad \frac{\Psi;\ \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\ \Phi \vdash t_2 : t}{\Psi;\ \Phi \vdash t_1\ t_2 : \lceil t' \rceil \cdot (\mathsf{id}_\Phi, t_2)} \qquad \frac{\Psi.i = [\Phi']t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}$$

$\Psi \vdash T : K$

$$\frac{\Psi;\ \Phi \vdash t : t'}{\Psi \vdash [\Phi]t : [\Phi]t'} \qquad \frac{\Psi \vdash \Phi,\ \Phi'\ \mathrm{wf}}{\Psi \vdash [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}}$$

$\Psi \vdash \Phi$ wf (sample)

$$\frac{\Psi \vdash \Phi\ \mathrm{wf} \qquad \Psi.i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash (\Phi,\ \phi_i)\ \mathrm{wf}}$$

(b) Extension variables: meta-variables and context variables

---

Subst. application: $t \cdot \sigma$

$$c \cdot \sigma = c \qquad f_{\mathbf{I}} \cdot \sigma = \sigma.\mathbf{I} \qquad b_i \cdot \sigma = b_i \qquad (\lambda(t_1).t_2) \cdot \sigma = \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) \qquad (t_1\ t_2) \cdot \sigma = (t_1 \cdot \sigma)\ (t_2 \cdot \sigma)$$

Ext. subst. application (sample)

$$\begin{array}{ll}(\mathbf{I},\ |\phi_i|) \cdot \sigma_\Psi = (\mathbf{I} \cdot \sigma_\Psi),\ |\Phi'| \text{ when } \sigma_\Psi.i = [\_]\Phi' & (X_i/\sigma) \cdot \sigma_\Psi = t \cdot (\sigma \cdot \sigma_\Psi) \text{ when } \sigma_\Psi.i = [\_]t \\ (\sigma,\ \mathbf{id}(\phi_i)) \cdot \sigma_\Psi = \sigma \cdot \sigma_\Psi,\ \mathsf{id}_{\sigma_\Psi.i} & (\Phi,\ \phi_i) \cdot \sigma_\Psi = \Phi \cdot \sigma_\Psi,\ \Phi' \text{ when } \sigma_\Psi.i = [\_]\Phi'\end{array}$$

$\Psi;\ \Phi \vdash \sigma : \Phi'$

$$\frac{}{\Psi;\ \Phi \vdash \bullet : \bullet} \qquad \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi;\ \Phi \vdash t : t' \cdot \sigma}{\Psi;\ \Phi \vdash (\sigma,\ t) : (\Phi',\ t')} \qquad \frac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\ \phi_i \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(\phi_i)) : (\Phi',\ \phi_i)}$$

$\Psi \vdash \sigma_\Psi : \Psi'$ (selected)

$$\frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : K \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\ T) : (\Psi',\ K)}$$

Subst. lemmas:

$$\frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi' \vdash \sigma : \Phi}{\Psi;\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma} \qquad \frac{\Psi;\ \Phi' \vdash \sigma : \Phi \qquad \Psi;\ \Phi'' \vdash \sigma' : \Phi'}{\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi} \qquad \frac{\Psi \vdash T : K \qquad \Psi' \vdash \sigma_\Psi : \Psi}{\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi}$$

(c) Substitutions over logical variables and extension variables

---

Syntax: $\Gamma ::= \bullet \mid \Gamma,\ x : \tau \mid \Gamma,\ x :_s \tau \mid \Gamma,\ \alpha : k \qquad e ::= \cdots \mid \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e'$

Limit ctx:

$$\begin{array}{lcl}\bullet|_{\mathrm{static}} & = & \bullet \\ (\Gamma,\ x :_s t)|_{\mathrm{static}} & = & \Gamma|_{\mathrm{static}},\ x : t \\ (\Gamma,\ x : t)|_{\mathrm{static}} & = & \Gamma|_{\mathrm{static}} \\ (\Gamma,\ \alpha : k)|_{\mathrm{static}} & = & \Gamma|_{\mathrm{static}}\end{array}$$

$\Psi;\ \Sigma;\ \Gamma \vdash e : \tau$ (part)

$$\frac{\bullet;\ \Sigma;\ \Gamma|_{\mathrm{static}} \vdash e : \tau \qquad \Psi;\ \Sigma;\ \Gamma, x :_s \tau \vdash e' : \tau}{\Psi;\ \Sigma;\ \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau} \qquad \frac{x :_s \tau \in \Gamma}{\Psi;\ \Sigma;\ \Gamma \vdash x : \tau}$$

Evaluation:

$$\begin{array}{l}v ::= \Lambda(K).e_d \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ v \mid () \mid \lambda x : \tau.e_d \mid (v,\ v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e_d \\ \mathcal{S} ::= \mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e' \mid \mathsf{letstatic}\ x\ =\ \mathcal{S}\ \mathsf{in}\ e' \mid \Lambda(K).\mathcal{S} \mid \lambda x : \tau.\mathcal{S} \mid \mathsf{unpack}\ e_d\ (.)x.(\mathcal{S}) \mid \mathsf{case}(e_d, x.\mathcal{S}, x.e_2) \\ \quad \mid \mathsf{case}(e_d, x.e_d, x.\mathcal{S}) \mid \Lambda\alpha : k.\mathcal{S} \mid \mathsf{fix}\ x : \tau.\mathcal{S} \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto \mathcal{S}) \mid \mathcal{E}_s[\mathcal{S}] \\ \mathcal{E}_s ::= \mathcal{E}_s\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ \mathcal{E}_s \mid \mathsf{unpack}\ \mathcal{E}_s\ (.)x.(e') \mid \mathcal{E}_s\ e' \mid e_d\ \mathcal{E}_s \mid (\mathcal{E}_s,\ e) \mid (e_d,\ \mathcal{E}_s) \mid \mathsf{proj}_i\ \mathcal{E}_s \mid \mathsf{inj}_i\ \mathcal{E}_s \\ \quad \mid \mathsf{case}(\mathcal{E}_s, x.e_1, x.e_2) \mid \mathsf{fold}\ \mathcal{E}_s \mid \mathsf{unfold}\ \mathcal{E}_s \mid \mathsf{ref}\ \mathcal{E}_s \mid \mathcal{E}_s := e' \mid e_d := \mathcal{E}_s \mid !\mathcal{E}_s \mid \mathcal{E}_s\ \tau \\ e_d ::= \text{all of } e \text{ except } \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' \qquad \mathcal{E} ::= \text{exactly as } \mathcal{E}_s \text{ with } \mathcal{E}_s \to \mathcal{E} \text{ and } e \to e_d\end{array}$$

Stage 1 op.sem.:

$$\frac{(\ \mu,\ e_d\ ) \longrightarrow (\ \mu',\ e'_d\ )}{(\ \mu,\ \mathcal{S}[e_d]\ ) \longrightarrow_s (\ \mu',\ \mathcal{S}[e'_d]\ )} \qquad (\ \mu,\ \mathcal{S}[\mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e]\ ) \longrightarrow_s (\ \mu,\ \mathcal{S}[e[v/x]]\ )$$

$$(\ \mu,\ \mathsf{letstatic}\ x\ =\ v\ \mathsf{in}\ e\ ) \longrightarrow_s (\ \mu,\ e[v/x]\ )$$

(d) Computational language: staging support

Figure 11. Main definitions in metatheory

a concrete context triggers a set of potentially complicated α-renamings, which a concrete representation makes explicit. We use a hybrid technique representing bound variables as deBruijn indices, and free variables as deBruijn levels. Our technique is a small departure from the named approach, requiring fewer extra annotations and lemmas than normal deBruijn indices. Also it identifies terms not only up to α-equivalence, but also up to extension of the context with new variables; this is why it is also used within the VeriML implementation.The two fundamental operations of this technique are freshening and binding, which are shown in Fig. 11a. Details can be found in section A of the appendix.

Extension variables. We extend the logic with support for meta-variables and context variables – we refer to both these sorts of variables as extension variables. A meta-variable X_i stands for a contextual term $T = [\Phi]t$, which packages a term together with the context it inhabits. Context variables ϕ_i stand for a context Φ, and are used to "weaken" parametric contexts in specific positions. Both kinds of variables are needed to support manipulation of open logical terms. Details of their definition and typing are shown in Fig. 11b. We use the same hybrid approach as above for representing these variables. A somewhat subtle aspect of this extension is that we generalize the deBruijn levels $\mathbf{I}$ used to index free variables, in order to deal effectively with parametric contexts.

Substitutions. The hybrid representation technique we use for variables renders simultaneous substitutions for all variables in scope as the most natural choice. In Fig. 11c, we show some example rules of how to apply a full simultaneous substitution σ to a term t, denoted as $t \cdot \sigma$. Similarly, we define full simultaneous substitutions σ_Ψ for extension contexts; defining their application has a very natural description, because of our variable representation technique. We prove a number of substitution lemmas which have simple statements, as shown in Fig. 11c. The proofs of these lemmas comprise the main effort required in proving the type-safety of a computational language such as the one we support, as they represent the point where computation specific to logical term manipulation takes place. Details can be found in section B of the appendix.

Computational language. We define an ML-style computational language that supports dependent functions and dependent pairs over contextual terms T, as well as pattern matching over them. Lack of space precludes us from including details here; full details can be found in section C of the appendix. A fairly complete ML calculus is supported, with mutable references and recursive types. Type safety is proved using standard techniques; its central point is extending the logic substitution lemmas to expressions and using them to prove progress and preservation of dependent functions and dependent pairs. This proof is modular with respect to the logic and other logics can easily be supported.

Pattern matching. Our metatheory includes many extensions in the pattern matching that is supported, as well as a new approach for dealing with typing patterns. We include support for pattern matching over contexts (e.g. to pick out hypotheses from the context) and for non-linear patterns. The allowed patterns are checked through a restriction of the usual typing rules $\Psi \vdash_p T : K$.

The essential idea behind our approach to pattern matching is to identify what the relevant variables in a typing derivation are. Since contexts are ordered, "removing" non-relevant variables amounts to replacing their definitions in the context with holes, which leads us to partial contexts $\widehat{\Psi}$. The corresponding notion of partial substitutions is denoted as $\widehat{\sigma_\Psi}$. Our main theorem about pattern matching can then be stated as:

Theorem 6.1 (Decidability of pattern matching) *If $\Psi \vdash_p T : K$, $\bullet \vdash_p T' : K$ and $\mathsf{relevant}(\Psi;\ \Phi \vdash T : K) = \widehat{\Psi}$, then either there exists a unique partial substitution $\widehat{\sigma_\Psi}$ such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$ and $T \cdot \widehat{\sigma_\Psi} = T'$, or no such substitution exists.*

Details are found in section D of the appendix.

Staging. Our development in this paper critically depends on the letstatic construct we presented earlier. It can be seen as a dual of the traditional box construct of Davies and Pfenning [1996]. Details of its typing and semantics are shown in Fig. 11d. We define a notion of "static evaluation contexts" $\mathcal{S}$, which enclose a hole of the form $\mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e$. They include normal evaluation contexts, as well as evaluation contexts under binding structures. We evaluate expressions e that include staging constructs using the $\longrightarrow_s$ relation; internally, this uses the normal evaluation rules, that are used in the second stage as well, for evaluating expressions which do not include other staging constructs. If stage-one evaluation is successful, we are left with a residual dynamic configuration (μ', e_d) which is then evaluated normally. We prove type-safety for stage-one evaluation; its statement follows.

Theorem 6.2 (Stage-one Type Safety) *If* $\bullet; \Sigma; \bullet \vdash e : \tau$ *then: either e is a dynamic expression* e_d*; or, for every store* μ *such that* $\vdash \mu : \Sigma$*, we have: either* $\mu, e \longrightarrow_s \mathsf{error}$*, or, there exists an* e'*, a new store typing* $\Sigma' \supseteq \Sigma$ *and a new store* μ' *such that:* $(\mu, e) \longrightarrow (\mu', e')$*;* $\vdash \mu' : \Sigma'$*; and* $\bullet; \Sigma'; \bullet \vdash e' : \tau$.

Details are found in section E of the appendix.

Collapsing extension variables. Last, we have proved the fact that under the conditions described in Sec. 5, it is possible to collapse a term t into a term t' which is typed under the empty extension variables context; a substitution σ with which we can regain the original term t exists. This suggests that whenever a proof object t for a specific proposition is required, an equivalent proof object that does not mention uninstantiated extension variables exists. Therefore, we can write an equivalent proof script producing the collapsed proof object instead, and evaluate that script statically. The statement of this theorem is the following:

Theorem 6.3 *If* $\Psi \vdash [\Phi]t : [\Phi]t_T$ *and* $\mathsf{collapsible}(\Psi \vdash [\Phi]t : [\Phi]t_T)$*, then there exist* Φ'*,* t'*,* t'_T *and* σ *such that* $\bullet \vdash \Phi'$ *wf,* $\bullet \vdash [\Phi']t' : [\Phi']t'_T$*,* $\Psi; \Phi \vdash \sigma : \Phi'$*,* $t' \cdot \sigma = t$ *and* $t'_T \cdot \sigma = t_T$.

The main idea behind the proof is to maintain a number of substitutions and their inverses: one to go from a general Ψ extension context into an "equivalent" Ψ' context, which includes only definitions of the form $[\Phi]t$, for a constant Φ context that uses no extension variables. Then, another substitution and its inverse are maintained to go from that extension variables context into the empty one; this is simpler, since terms typed under Ψ' are already essentially free of metavariables. The computational content within the proof amounts to a procedure for transforming proof scripts inside tactics into static proof scripts. Details are found in section F of the appendix.

7. Implementation

We have completed a prototype implementation of the VeriML language, as described in this paper, that supports all of our claims. We have built on our existing prototype [Stampoulis and Shao 2010] and have added an extensive set of new features and improvements. The prototype is written in OCaml and is about 6k lines of code. Using the prototype we have implemented a number of examples, that are about 1.5k lines of code. Readers are encouraged to download and try the prototype from `http://flint.cs.yale.edu/publications/supc.html`.

New features. We have implemented the new features we have described so far: context matching, non-linear patterns, proof-erasure semantics, staging, and inferencing for logical and computational terms. Proof-erasure semantics are utilized only if requested by a per-function flag, enabling us to selectively "trust" tactics. The staging construct we support is more akin to the $\langle\cdot\rangle_{\text{static}}$ form described as syntactic sugar in Sec. 5, and it is able to infer the collapsing substitutions that are needed, following the approach used in our metatheory.

Changes. We have also changed quite a number of things in the prototype and improved many of its aspects. A central change, mediated by our new treatment of the conversion rule, was to modify the used logic in

order to use the explicit equality approach; the existing prototype used the λHOL_c logic. We also switched the variable representation to the hybrid deBruijn levels-deBruijn indices technique we described, which enabled us to implement subtyping based on context subsumption. Also, we have adapted the typing rules of the pattern matching construct in order to support refining the environment based on the current branch.

Examples implemented. We have implemented a number of examples to support our claims. First, we have written the type-safe conversion check routine for $\beta\mathbb{N}$, and extended it to support congruence closure based on equalities in the context. Proofs of this latter tactic are constructed automatically through static proof scripts, using a naive rewriter that is non-terminating in the general case. We have also completed proofs for theorems of arithmetic for the properties of addition and multiplication, and used them to write an arithmetic simplification tactic. All of the theorems are proved by making essential use of existing conversion rules, and are immediately added into new conversion rules, leading to a compact and clean development style. The resulting code does not need to make use of translation validation or proof by reflection, which are typically used to implement similar tactics in existing proof assistants.

Towards a practical proof assistant. In order to facilitate practical proof and program construction in VeriML, we introduced some features to support surface syntax, enabling users to omit most details about the environments of contextual terms and the substitutions used with meta-variables. This syntax follows the style of Delphin, assuming an ambient logical variable environment which is extended through a construct denoted as $\nu x : t.e$. Still, the full power of contextual modal type theory is available, which is crucial in order to change what the current ambient environment is, used, as we saw earlier, for static calls to tactics. In general the surface syntax leads to much more concise and readable code.

Last, we introduced syntax support for calls to tactics, enabling users to write proof expressions that look very similar to proof scripts in current proof assistants. We developed a rudimentary ProofGeneral mode for VeriML, that enables us to call the VeriML type-checker and interpreter for parts of source files. By adding holes to our sources, we can be informed by the type inference mechanism about their expected types. Those types correspond to what the current "proof state" is at that point. Therefore, a possible workflow for developing tactics or proofs, is writing the known parts, inserting holes in missing points to know what remains to be proved, and calling the typechecker to get the proof state information. This workflow corresponds closely to the interactive proof development support in proof assistants like Coq and Isabelle, but generalizes it to the case of tactics as well.

8. Related work

There is a large body of work that is related to the ideas we have presented here.

Techniques for robust proof development. There have been multiple proposals for making proof development inside existing proof assistants more robust. A well-known technique is *proof-by-reflection* [Boutin 1997]: writing total and certified decision procedures within the functional language contained in a logic like CIC. A recently introduced technique is *automation through canonical structures* [Gonthier et al. 2011]: the resolution mechanism for finding instances of canonical structures (a generalization of type classes) is cleverly utilized in order to program automation procedures for specific classes of propositions. We view both approaches as somewhat similar, as both are based in cleverly exploiting static "interpreters" that are available in a modern proof assistant: the partial evaluator within the conversion rule in the former case; the unification algorithm within instance discovery in the latter case.

Our approach can thus be seen as similar, but also as a generalization of these approaches, since a general-purpose programming model is supported. Therefore, users do not have to adapt to a specific programming style for writing automation code, but can rather use a familiar functional language. Proof-by-reflection could perhaps be used to support the same kind of extensions to the conversion rule; still, this would require reflecting

a large part of the logic in itself, through a prohibitively complicated encoding. Both techniques are applicable to our setting as well and could be used to provide benefits to large developments within our language.

The style advocated in Chlipala [2011] (and elsewhere) suggests that proper proof engineering entails developing sophisticated automation tactics in a modular style, and extending their power by adding proved lemmas as hints. We are largely inspired by this approach, and believe that our introduction of the extensible conversion rule and static checking of tactics can significantly benefit it. We demonstrate similar ideas in layering conversion tactics.

Traditional proof assistants. There are many parallels of our work with the *LCF family of proof assistants*, like HOL4 [Slind and Norrish 2008] and HOL-Light [Harrison 1996], which have served as inspiration. First, the foundational logic that we use is similar. Also, our use of a dedicated ML-like programming language to program tactics and proof scripts is similar to the approach taken by HOL4 and HOL-Light. Last, the fact that no proof objects need to be generated is shared. Still, checking a proof script in HOL requires evaluating it fully. Using our approach, we can selectively evaluate parts of proof scripts; we focus on conversion-like tactics, but we are not limited inherrently to those. This is only possible because our proof scripts carry proof state information within their types. Similarly, proof scripts contained within LCF tactics cannot be evaluated statically, so it is impossible to establish their validity upon tactic definition. It is possible to do a transformation similar to ours manually (lifting proof scripts into auxiliary lemmas that are proved prior to the tactic), but the lack of type information means that many more details need to be provided.

The Coq proof assistant [Barras et al. 2010] is another obvious point of reference for our work. We will focus on the conversion rule that CIC, its accompanying logic, supports – the same problems with respect to proof scripts and tactics that we described in the LCF case also apply for Coq. The conversion rule, which identifies computationally equivalent propositions, coupled with the rich type universe available, opens up many possibilities for constructing small and efficiently checkable proof objects. The implementation of the conversion rule needs to be part of the trusted base of the proof assistant. Also, the fact that the conversion check is built-in to the proof assistant makes the supported equivalence rigid and non-extensible by frequently used decision procedures.

There is a large body of work that aims to extend the conversion rule to arbitrary confluent rewrite systems (e.g. Blanqui et al. [1999]) and to include decision procedures [Strub 2010]. These approaches assume some small or larger addition to the trusted base, and extend the already complex metatheory of Coq. Furthermore, the NuPRL proof assistant [Constable et al. 1986] is based on extensional type theory which includes an extensional conversion rule. This enables complex decision procedures to be part of conversion; but it results in a very large trusted base. We show how, for a subset of these type theories, the conversion check can be recovered outside the trusted base. It can be extended with arbitrarily complex new tactics, written in a familiar programming style, without any metatheoretic additions and without hurting the soundness of the logic. The question of whether these type theories can be supported in full remains as future work, but as far as we know, there is no inherrent limitation to our approach.

Dependently-typed programming. The large body of work on dependently-typed languages has close parallels to our work. Out of the multitude of proposals, we consider the Russell framework [Sozeau 2006] as the current state-of-the-art, because of its high expressivity and automation in discharging proof obligations. In our setting, we can view dependently-typed programming as a specific case of tactics producing complex data types that include proof objects. Static proof scripts can be leveraged to support expressivity similar to the Russell framework. Furthermore, our approach opens up a new intriguing possibility: dependently-typed programs whose obligations are discharged statically and automatically, through code written within the same language.

Last, we have been largely inspired by the work on languages like Beluga [Pientka and Dunfield 2008] and Delphin [Poswolsky and Schürmann 2008], and build on our previous work on VeriML [Stampoulis and Shao

2010]. We investigate how to leverage type-safe tactics, as well as a number of new constructs we introduce, so as to offer an extensible notion of proof checking. Also, we address the issue of statically checking the proof scripts contained within tactics written in VeriML. As far as we know, our development is the first time languages such as these have been demonstrated to provide a workflow similar to interactive proof assistants.

Acknowledgments

We thank anonymous referees for their suggestions and comments on an earlier version of this paper. This research is based on work supported in part by DARPA CRASH grant FA8750-10-2-0254 and NSF grants CCF-0811665, CNS-0910670, and CNS 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

H.P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Sci. Pub. B.V., 1999.

B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, et al. The Coq proof assistant reference manual (version 8.3), 2010.

F. Blanqui, J.P. Jouannaud, and M. Okada. The calculus of algebraic constructions. In *Rewriting Techniques and Applications*, pages 671–671. Springer, 1999.

F. Blanqui, J.P. Jouannaud, and P.Y. Strub. A calculus of congruent constructions. *Unpublished draft*, 2005.

S. Boutin. Using reflection to build efficient and certified decision procedures. *Lecture Notes in Computer Science*, 1281: 515–529, 1997.

A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 2011 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2011.

R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

R. Davies and F. Pfenning. A modal analysis of staged computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 258–270. ACM, 1996.

G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, pages 163–175. ACM, 2011.

J. Harrison. HOL Light: A tutorial introduction. *Lecture Notes in Computer Science*, pages 265–269, 1996.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

T. Nipkow, L.C. Paulson, and M. Wenzel. Isabelle/HOL : A Proof Assistant for Higher-Order Logic, volume 2283 of LNCS, 2002.

B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 163–173. ACM New York, NY, USA, 2008.

A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. *Lecture Notes in Computer Science*, 4960:93, 2008.

V. Siles and H. Herbelin. Equality is typable in semi-full pure type systems. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 21–30. IEEE, 2010.

K. Slind and M. Norrish. A brief overview of HOL4. *Theorem Proving in Higher Order Logics*, pages 28–32, 2008.

M. Sozeau. Subset coercions in coq. In *Proceedings of the 2006 International Conference on Types for Proofs and Programs*, pages 237–252. Springer-Verlag, 2006.

A. Stampoulis and Z. Shao. VeriML: Typed computation of logical terms inside a language with effects. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 333–344. ACM, 2010.

P.Y. Strub. Coq modulo theory. In *Proceedings of the 24th International Conference on Computer Science Logic*, pages 529–543. Springer-Verlag, 2010.

Appendices

A. The logic λHOL_c

Definition A.1 (Syntax of the language) *The syntax of the logic language is given below.*

$$
\begin{array}{rl}
t ::= & s \mid c \mid f_i \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \mid \mathsf{conv}\ t\ t \mid \mathsf{refl}\ t \mid \mathsf{symm}\ t \mid \mathsf{trans}\ t_1\ t_2 \mid \mathsf{congapp}\ t_1\ t_2 \\
& \mid \mathsf{congimpl}\ t_1\ t_2 \mid \mathsf{conglam}\ t \mid \mathsf{congpi}\ t \mid \mathsf{beta}\ t_1\ t_2 \\
s ::= & \mathsf{Prop} \mid \mathsf{Type} \mid \mathsf{Type}' \\
\Phi ::= & \bullet \mid \Phi,\ t \\
\sigma ::= & \bullet \mid t \\
\Sigma ::= & \bullet \mid \Sigma,\ c : t
\end{array}
$$

We use f_i to denote the i-th free variable in the current environment and b_i for the bound variable with deBruijn index i. The benefit of this approach is that the representation of terms is unique both up to α-equivalence and up to extensions of the current free variables context.

Definition A.2 (Context length and access) *Getting the length of a context, and an element out of a context, are defined as follows. In the case of element access, we assume that $i < |\Phi|$.*

$\boxed{|\Phi|}$

$$
\begin{array}{lcl}
|\bullet| & = & 0 \\
|\Phi,t| & = & |\Phi| + 1
\end{array}
$$

$\boxed{\Phi.i}$

$$
\begin{array}{lcl}
(\Phi,t).|\Phi| & = & t \\
(\Phi,t).i & = & \Phi.i
\end{array}
$$

Definition A.3 (Substitution length) *Getting the length of a substitution is defined as follows.*

$$
\begin{array}{lcl}
|\bullet| & = & 0 \\
|\sigma,t| & = & |\sigma| + 1
\end{array}
$$

Definition A.4 (Substitution access) *The operation of accessing the i-th term out of a substitution is defined as follows. We assume that $i < |\sigma|$.*

$$
\begin{array}{lcl}
(\sigma,t).|\sigma| & = & t \\
(\sigma,t).i & = & \sigma.i
\end{array}
$$

Definition A.5 (Substitution application) *The operation of applying a substitution is defined as follows.*

$\boxed{t \cdot \sigma}$

$$
\begin{array}{lcl}
s \cdot \sigma & = & s \\
c \cdot \sigma & = & c \\
f_i \cdot \sigma & = & \sigma.i \\
b_i \cdot \sigma & = & b_i \\
(\lambda(t_1).t_2) \cdot \sigma & = & \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) \\
(t_1\ t_2) \cdot \sigma & = & (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
(\Pi(t_1).t_2) \cdot \sigma & = & \Pi(t_1 \cdot \sigma).(t_2 \cdot \sigma) \\
(t_1 = t_2) \cdot \sigma & = & (t_1 \cdot \sigma) = (t_2 \cdot \sigma) \\
(\mathsf{conv}\ t_1\ t_2) \cdot \sigma & = & \mathsf{conv}\ (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
(\mathsf{refl}\ t) \cdot \sigma & = & \mathsf{refl}\ (t \cdot \sigma) \\
(\mathsf{symm}\ t) \cdot \sigma & = & \mathsf{symm}\ (t \cdot \sigma) \\
(\mathsf{trans}\ t_1\ t_2) \cdot \sigma & = & \mathsf{trans}\ (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
(\mathsf{congapp}\ t_1\ t_2) \cdot \sigma & = & \mathsf{congapp}\ (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
(\mathsf{congimpl}\ t_1\ t_2) \cdot \sigma & = & \mathsf{congimpl}\ (t_1 \cdot \sigma)\ (t_2 \cdot \sigma) \\
(\mathsf{conglam}\ t) \cdot \sigma & = & \mathsf{conglam}\ (t \cdot \sigma) \\
(\mathsf{congpi}\ t) \cdot \sigma & = & \mathsf{congpi}\ (t \cdot \sigma) \\
(\mathsf{beta}\ t_1\ t_2) \cdot \sigma & = & \mathsf{beta}\ (t_1 \cdot \sigma)\ (t_2 \cdot \sigma)
\end{array}
$$

$\boxed{\sigma' \cdot \sigma}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma & = & \bullet \\
(\sigma', t) \cdot \sigma & = & (\sigma' \cdot \sigma), (t \cdot \sigma)
\end{array}
$$

Definition A.6 (Identity substitution) *The identity substitution is defined as follows.*

$$
\begin{array}{lcl}
\mathsf{id}_\bullet & = & \bullet \\
\mathsf{id}_{\Phi,\, t} & = & \mathsf{id}_\Phi,\ f_{|\Phi|}
\end{array}
$$

Definition A.7 *Free and bound variable limits for terms are defined as follows.*

$\boxed{t <^f n}$

$$
\begin{array}{lcl}
s <^f n & & \\
c <^f n & & \\
f_i <^f n & \Leftarrow & n > i \\
b_i <^f n & & \\
(\lambda(t_1).t_2) <^f n & \Leftarrow & t_1 <^f n \wedge t_2 <^f n \\
t_1\ t_2 <^f n & \Leftarrow & t_1 <^f n \wedge t_2 <^f n \\
 & \cdots &
\end{array}
$$

$\boxed{t <^b n}$

$$
\begin{array}{lcl}
s <^b n & & \\
c <^b n & & \\
f_i <^b n & & \\
b_i <^b n & \Leftarrow & n > i \\
(\lambda(t_1).t_2) <^b n & \Leftarrow & t_1 <^b n \wedge t_2 <^b n + 1 \\
t_1\ t_2 <^b n & \Leftarrow & t_1 <^b n \wedge t_2 <^b n \\
 & \cdots &
\end{array}
$$

Definition A.8 *Free and bound variable limits for substitutions are defined as follows.*

$\boxed{\sigma <^f n}$

$$\begin{array}{l} \bullet <^f n \\ (\sigma, t) <^f n \quad \Leftarrow \quad \sigma <^f n \wedge t <^f n \end{array}$$

$\boxed{\sigma <^b n}$

$$\begin{array}{l} \bullet <^b n \\ (\sigma, t) <^b n \quad \Leftarrow \quad \sigma <^b n \wedge t <^b n \end{array}$$

Definition A.9 (Freshening) *Freshening a term is defined as follows. We assume that $t <^f m$ and $t <^b n+1$.*

$\boxed{\lceil t \rceil^n_m}$

$$\begin{array}{lcl}
\lceil s \rceil & = & s \\
\lceil c \rceil & = & c \\
\lceil f_i \rceil & = & f_i \\
\lceil b_n \rceil^n_m & = & f_m \\
\lceil b_i \rceil^n & = & b_i \text{ when } i < n \\
\lceil (\lambda(t_1).t_2) \rceil^n & = & \lambda(\lceil t_1 \rceil^n).\lceil t_2 \rceil^{n+1} \\
\lceil t_1\ t_2 \rceil & = & \lceil t_1 \rceil\ \lceil t_2 \rceil \\
\lceil \Pi(t_1).t_2) \rceil^n & = & \Pi(\lceil t_1 \rceil^n).(\lceil t_2 \rceil^{n+1}) \\
\lceil t_1 = t_2 \rceil & = & \lceil t_1 \rceil = \lceil t_2 \rceil \\
\lceil \mathsf{conv}\ t_1\ t_2 \rceil & = & \mathsf{conv}\ \lceil t_1 \rceil\ \lceil t_2 \rceil \\
\lceil \mathsf{refl}\ t \rceil & = & \mathsf{refl}\ \lceil t \rceil \\
\lceil \mathsf{symm}\ t \rceil & = & \mathsf{symm}\ \lceil t \rceil \\
\lceil \mathsf{trans}\ t_1\ t_2 \rceil & = & \mathsf{trans}\ \lceil t_1 \rceil\ \lceil t_2 \rceil \\
\lceil \mathsf{congapp}\ t_1\ t_2 \rceil & = & \mathsf{congapp}\ \lceil t_1 \rceil\ \lceil t_2 \rceil \\
\lceil \mathsf{congimpl}\ t_1\ t_2 \rceil & = & \mathsf{congimpl}\ \lceil t_1 \rceil\ \lceil t_2 \rceil \\
\lceil \mathsf{conglam}\ t \rceil & = & \mathsf{conglam}\ \lceil t \rceil \\
\lceil \mathsf{congpi}\ t \rceil & = & \mathsf{congpi}\ \lceil t \rceil \\
\lceil \mathsf{beta}\ t_1\ t_2 \rceil & = & \mathsf{beta}\ \lceil t_1 \rceil\ \lceil t_2 \rceil
\end{array}$$

Definition A.10 (Binding) *Binding a term is defined as follows. We assume that $t <^f m$ and $t <^b n$.*

$\boxed{\lfloor t \rfloor_m^n}$

$$
\begin{array}{lcl}
\lfloor s \rfloor & = & s \\
\lfloor c \rfloor & = & c \\
\lfloor f_{m-1} \rfloor_m^n & = & b_n \\
\lfloor f_i \rfloor_m^n & = & f_i \text{ when } i < m-1 \\
\lfloor b_i \rfloor & = & b_i \\
\lfloor (\lambda(t_1).t_2) \rfloor & = & \lambda(\lfloor t_1 \rfloor^n).\, \lfloor t_2 \rfloor^{n+1} \\
\lfloor t_1\ t_2 \rfloor & = & \lfloor t_1 \rfloor^n\ \lfloor t_2 \rfloor^n \\
\lfloor \Pi(t_1).t_2) \rfloor & = & \Pi(\lfloor t_1 \rfloor^n).\, \lfloor t_2 \rfloor^{n+1} \\
\lfloor t_1 = t_2 \rfloor & = & \lfloor t_1 \rfloor = \lfloor t_2 \rfloor \\
\lfloor \mathsf{conv}\ t_1\ t_2 \rfloor & = & \mathsf{conv}\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \mathsf{refl}\ t \rfloor & = & \mathsf{refl}\ \lfloor t \rfloor \\
\lfloor \mathsf{symm}\ t \rfloor & = & \mathsf{symm}\ \lfloor t \rfloor \\
\lfloor \mathsf{trans}\ t_1\ t_2 \rfloor & = & \mathsf{trans}\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \mathsf{congapp}\ t_1\ t_2 \rfloor & = & \mathsf{congapp}\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \mathsf{congimpl}\ t_1\ t_2 \rfloor & = & \mathsf{congimpl}\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \mathsf{conglam}\ t \rfloor & = & \mathsf{conglam}\ \lfloor t \rfloor \\
\lfloor \mathsf{congpi}\ t \rfloor & = & \mathsf{congpi}\ \lfloor t \rfloor \\
\lfloor \mathsf{beta}\ t_1\ t_2 \rfloor & = & \mathsf{beta}\ \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor
\end{array}
$$

Definition A.11 (Typing) *The typing rules are defined as follows.*

$\boxed{\vdash \Sigma\ \mathrm{wf}}$

$$\vdash \bullet\ \mathrm{wf} \qquad \frac{\vdash \Sigma\ \mathrm{wf} \qquad \bullet \vdash_\Sigma t : s \qquad (c : \_) \notin \Sigma}{\vdash \Sigma,\, c : t\ \mathrm{wf}}$$

$\boxed{\vdash_\Sigma \Phi\ \mathrm{wf}}$

$$\vdash \bullet\ \mathrm{wf} \qquad \frac{\vdash \Phi\ \mathrm{wf} \qquad \Phi \vdash t : s}{\vdash \Phi,\, t\ \mathrm{wf}}$$

$\boxed{\Phi \vdash_\Sigma t : t'}$

$$\frac{c : t \in \Sigma}{\Phi \vdash_\Sigma c : t} \qquad \frac{\Phi.i = t}{\Phi \vdash f_i : t} \qquad \frac{(s,s') \in \mathcal{A}}{\Phi \vdash s : s'} \qquad \frac{\Phi \vdash t_1 : s \qquad \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s,s',s'') \in \mathcal{R}}{\Phi \vdash \Pi(t_1).t_2 : s''}$$

$$\frac{\Phi \vdash t_1 : s \qquad \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Phi \vdash \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|+1} : s'}{\Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|+1}} \qquad \frac{\Phi \vdash t_1 : \Pi(t).t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}$$

$$\frac{\Phi \vdash t_1 : t \qquad \Phi \vdash t_2 : t \qquad \Phi \vdash t : \mathsf{Type}}{\Phi \vdash t_1 = t_2 : \mathsf{Prop}}$$

$$\frac{\Phi \vdash t : t_1 \qquad \Phi \vdash t_1 : \mathsf{Prop} \qquad \Phi \vdash t' : t_1 = t_2}{\Phi \vdash \mathsf{conv}\ t\ t' : t_2} \qquad \frac{\Phi \vdash t_1 : t \qquad \Phi \vdash t_1 = t_1 : \mathsf{Prop}}{\Phi \vdash \mathsf{refl}\ t_1 : t_1 = t_1} \qquad \frac{\Phi \vdash t_a : t_1 = t_2}{\Phi \vdash \mathsf{symm}\ t_a : t_2 = t_1}$$

$$\frac{\Phi \vdash t_a : t_1 = t_2 \qquad \Phi \vdash t_b : t_2 = t_3}{\Phi \vdash \mathsf{trans}\ t_a\ t_b : t_1 = t_3}$$

$$\frac{\Phi \vdash t_a : M_1 = M_2 \qquad \Phi \vdash M_1 : A \to B \qquad \Phi \vdash t_b : N_1 = N_2 \qquad \Phi \vdash N_1 : A}{\Phi \vdash \mathsf{congapp}\ t_a\ t_b : M_1\ N_1 = M_2\ N_2}$$

$$\frac{\Phi \vdash t_a : A_1 = A_2 \qquad \Phi, A_1 \vdash \lceil t_b \rceil : B_1 = B_2 \qquad \Phi \vdash A_1 : \mathsf{Prop} \qquad \Phi, A_1 \vdash \lceil B_1 \rceil : \mathsf{Prop}}{\Phi \vdash \mathsf{congimpl}\ t_a\ (\lambda(A_1).t_b) : \Pi(A_1).\lfloor B_1 \rfloor = \Pi(A_2).\lfloor B_2 \rfloor}$$

$$\frac{\Phi,\ A \vdash \lceil t_b \rceil : B = B' \qquad \Phi \vdash \Pi(A).\lfloor B \rfloor = \Pi(A).\lfloor B' \rfloor : \mathsf{Prop}}{\Phi \vdash \mathsf{congpi}\ (\lambda(A).t_b) : \Pi(A).\lfloor B \rfloor = \Pi(A).\lfloor B' \rfloor}$$

$$\frac{\Phi,\ A \vdash \lceil t_b \rceil : B_1 = B_2 \qquad \Phi \vdash \lambda(A).\lfloor B_1 \rfloor = \lambda(A).\lfloor B_2 \rfloor : \mathsf{Prop}}{\Phi \vdash \mathsf{conglam}\ (\lambda(A).t_b) : \lambda(A).\lfloor B_1 \rfloor = \lambda(A).\lfloor B_2 \rfloor}$$

$$\frac{\Phi \vdash \lambda(A).M : A \to B \qquad \Phi \vdash N : A \qquad \Phi \vdash A \to B : \mathsf{Type}}{\Phi \vdash \mathsf{beta}\ (\lambda(A).M)\ N : (\lambda(A).M)\ N = \lceil M \rceil \cdot (\mathsf{id}_\Phi, N)}$$

$\boxed{\Phi \vdash \sigma : \Phi'}$

$$\frac{\vdash \Phi\ \mathrm{wf}}{\Phi \vdash \bullet : \bullet} \qquad \frac{\Phi \vdash \sigma : \Phi' \qquad \Phi \vdash t : t' \cdot \sigma}{\Phi \vdash \sigma, t : (\Phi', t')}$$

Lemma A.12 *If $t <^f m$ and $|\Phi| = m$ then $t \cdot id_\Phi = t$.*

Trivial by induction on $t <^f m$. The interesting case is $f_i \cdot \mathsf{id}_\Phi = f_i$. This is simple to prove by induction on Φ.

Lemma A.13 *If $\sigma <^f m$ then $\sigma \cdot id_m = \sigma$.*

By induction on σ and use of lemma A.12.

Lemma A.14 *If $\Phi \vdash t : t'$ then $t <^f |\Phi|$ and $t <^b 0$.*

Trivial by induction on the typing derivation $\Phi \vdash t : t'$ (and use of implicit assumptions for $\lceil t \rceil$).

Lemma A.15 *If $\vdash \Phi$ wf then for any Φ' and $t_{1\cdots n}$ such that $\Phi' = \Phi,\ t_1,\ t_2, \cdots, t_n$ and $\vdash \Phi'$ wf, we have that $\Phi' \vdash id_\Phi : \Phi$.*

By induction on Φ.
In case $\Phi = \bullet$, trivial.
In case $\Phi = \Phi'',\ t'$, then by induction hypothesis we have for all proper extensions of Φ'' $\Phi'', t_1, \cdots, t_n \vdash \mathsf{id}_{\Phi''} : \Phi''$.

We now need to prove that for all proper extensions of Φ'', t' we have Φ'', t', t_1, $\cdots$, $t_n \vdash \mathsf{id}_{\Phi'', t'} : (\Phi'', t')$.
From the inductive hypothesis we get that Φ'', t', t_1, $\cdots$, $t_n \vdash \mathsf{id}_{\Phi''} : \Phi''$. We also have that $\Phi'' \vdash t' : s$ by inversion of the well-formedness of Φ.
Thus by A.14, we get that $t' <^f |\Phi''|$.
Furthermore by A.12 we get that $t' \cdot \mathsf{id}_{\Phi''} = t'$.
Thus we have $\Phi'', t', t_1, \cdots, t_n \vdash f_{|\Phi''|} : t' \cdot \mathsf{id}_{\Phi''}$.
Thus by applying the appropriate substitution typing rule, we get that $\Phi'', t', t_1, \cdots, t_n \vdash (\mathsf{id}_{\Phi''}, f_{|\Phi''|}) : (\Phi'', t')$, which is exactly the desired result.

Lemma A.16 *If* $\Phi \vdash \sigma : \Phi'$ *then* $\sigma <^f |\Phi|$, $\sigma <^b 0$ *and* $|\sigma| = |\Phi'|$.

Trivial by induction on the typing derivation for σ, and use of lemma A.14.

Lemma A.17 *If* $\vdash \Phi$ *wf and* $|\Phi| = n$ *then for all* $i < n$, $\Phi.i <^f i$.

Trivial by induction on the well-formedness derivation for Φ and use of lemma A.14.

Lemma A.18 *If* $t <^f m$, $|\sigma| = m$ *and* $t \cdot \sigma = t'$ *then* $t \cdot (\sigma, t_1, t_2, \cdots, t_n) = t'$.

Trivial by induction on $t <^f m$.

Lemma A.19 *If* $\sigma <^f m$, $|\sigma'| = m$ *and* $\sigma \cdot \sigma' = \sigma_r$ *then* $\sigma \cdot (\sigma', t_1, t_2, \cdots, t_n) = \sigma_r$.

Trivial by induction on σ, and use of the lemma A.18.

Lemma A.20 *If* $\vdash \Phi$ *wf,* $\Phi.i = t$ *and* $\Phi' \vdash \sigma : \Phi$, *then* $\Phi' \vdash \sigma.i : t \cdot \sigma$.

Induction on the derivation of typing for σ.
In the case where $\sigma = \bullet$, the (implicit) assumption that $i < |\Phi|$ obviously does not hold, so the case is impossible.
In the case where $\sigma = \sigma', t'$, we split cases on whether $i = |\Phi| - 1$ or not.
If it is, then the typing rule gives us the desired directly.
If it is not, the inductive hypothesis gives us the result $\Phi' \vdash \sigma'.i : t \cdot \sigma'$. Now from lemma A.17 we have that $\Phi.i <^f i$. We can now apply lemma A.18 to get $t \cdot \sigma' = t \cdot (\sigma', t') = t \cdot \sigma$, proving the desired.

Lemma A.21 *If* $t <^f m$, $t <^b n+1$, $\sigma <^f m'$ *and* $|\sigma| = m$ *then* $\lceil t \cdot \sigma \rceil^n_{m'} = \lceil t \rceil^n_m \cdot (\sigma, f_{m'})$.

By structural induction on t.
Cases $t = s$ and $t = c$ are trivial.
When $t = f_i$, we have $i < m$ thus both sides will be equal to $\sigma.i$.
When $t = b_i$, we split cases on whether $i = n$ or $i < n$.
If $i = n$, then the left-hand side becomes $\lceil b_n \cdot \sigma \rceil^n_{m'} = \lceil b_n \rceil^n_{m'} = f_{m'}$.
The right-hand side becomes $\lceil b_n \rceil^n_m \cdot (\sigma, f_{m'}) = f_m \cdot (\sigma, f_{m'}) = f_{m'}$.
When $i < n$ it is trivial to see that both sides are equal to bi.
In the case where $t = \lambda(t_1).(t_2)$, we prove the result trivially using the induction hypothesis.
The subtlety for t_2 is that the inductive hypothesis is applied for $n = n+1$, which is possible because from the definition of $\cdot <^b \cdot$ we have that $t_2 <^b (n+1)+1$.

Lemma A.22 *If* $t <^f m+1$, $t <^b n$, $\sigma <^f m'$ *and* $|\sigma| = m$ *then* $\lfloor t \cdot (\sigma, f_{m'}) \rfloor^n_{m'+1} = \lfloor t \rfloor^n_{m+1} \cdot \sigma$.

By structural induction on t. Cases $t = s$ and $t = c$ are trivial. When $t = f_i$, we split cases on whether $i = m$ or $i < m$. If $i = m$, then the left hand side becomes: $\lfloor f_m \cdot (\sigma, f_{m'}) \rfloor_{m'+1}^{n} = \lfloor f_{m'} \rfloor_{m'+1}^{n} = b_n$. The right hand side becomes: $\lfloor f_m \rfloor_{m+1}^{n} \cdot \sigma = b_n \cdot \sigma = b_n$. In case $i < m$, both sides are trivially equal to $\sigma.i$. When $t = b_i$, both sides are trivially equal to b_i. When $t = \lambda(t_1).t_2$, the result follows directly from the inductive hypothesis for t_1 and t_2, and the definitions of $\cdot$ and $\lfloor \cdot \rfloor$.

Lemma A.23 *If $t <^f m$, $|\sigma| = m$, $\sigma <^f m'$ and $|\sigma'| = m'$ then $(t \cdot \sigma) \cdot \sigma' = t \cdot (\sigma \cdot \sigma')$.*

Trivial induction, with the only interesting case where $t = f_i$. The left hand side becomes $(f_i \cdot \sigma) \cdot \sigma' = (\sigma.i) \cdot \sigma'$. The right hand side becomes $f_i \cdot (\sigma \cdot \sigma') = (\sigma \cdot \sigma').i = (\sigma.i) \cdot \sigma'$.

Lemma A.24 *If $|\sigma| = m$ and $|\Phi| = m$ then $id_\Phi \cdot \sigma = \sigma$.*

Trivial by induction on Φ.

Lemma A.25 *If $\lceil t \rceil_m^n = \lceil t' \rceil_m^n$ then $t = t'$.*

By induction on the structure of t. In each case we perform induction on t' as well. The only interesting case is when $t = f_i$ and $t' = b_n$. We have that $\lceil t' \rceil = f_m$; so it could be that $i = m$. This is avoided from the implicit assumption that $t <^f m$ (that is required to apply freshening).

The main substitution theorem that we are proving is the following.

Theorem A.26 (Substitution)
If $\Phi \vdash t : t'$ and $\Phi' \vdash \sigma : \Phi$ then $\Phi' \vdash t \cdot \sigma : t' \cdot \sigma'$.

By structural induction on the typing derivation for t.

Case $\dfrac{c : t \in \Sigma}{\Phi \vdash_\Sigma c : t}$ $\triangleright$
By applying the same typing rule we get that $\Phi' \vdash c : t$. By inversion of the well-formedness of Σ, we get that $\bullet \vdash t : t'$. Thus from lemma A.14 we get that $t <^f 0$ and from lemma A.18 we get that $t \cdot \sigma = t$. Considering also that $c \cdot \sigma = c$, the derivation $\Phi' \vdash c : t$ proves the desired.

Case $\dfrac{\Phi.i = t}{\Phi \vdash f_i : t}$ $\triangleright$
We have that $f_i \cdot \sigma = \sigma.i$. Directly using lemma A.20 we get that $\Phi' \vdash \sigma.i : t \cdot \sigma$.

Case $\dfrac{(s, s') \in \mathcal{A}}{\Phi \vdash s : s'}$ $\triangleright$
Trivial by application of the same rule and the definition of $\cdot$.

Case $\dfrac{\Phi \vdash t_1 : s \qquad \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Phi \vdash \Pi(t_1).t_2 : s''}$ $\triangleright$
By induction hypothesis for t_1 we get: $\Phi' \vdash t_1 \cdot \sigma : s$.
By induction hypothesis for $\Phi, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s'$ and $\Phi', t_1 \cdot \sigma \vdash (\sigma, f_{|\Phi'|}) : (\Phi, t_1)$ we get: $\Phi', t_1 \cdot \sigma \vdash \lceil t_2 \rceil_{|\Phi|} \cdot (\sigma, f_{|\Phi'|}) :$

$s' \cdot (\sigma, f_{|\Phi'|})$.
We have $s' = s' \cdot (\sigma, f_{|\Phi'|})$ trivially.
Also by the lemma A.21, $\lceil t_2 \rceil_{|\Phi|} \cdot (\sigma, f_{|\Phi'|}) = \lceil t_2 \cdot \sigma \rceil_{|\Phi'|}$.
Thus by application of the same typing rule we get $\Phi' \vdash \Pi(t_1 \cdot \sigma).(t_2 \cdot \sigma) : s''$ which is the desired, since $(\Pi(t_1).t_2) \cdot \sigma = \Pi(t_1 \cdot \sigma).(t_2 \cdot \sigma)$.

Case $\dfrac{\Phi \vdash t_1 : s \qquad \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Phi \vdash \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|+1} : s'}{\Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|+1}}$ $\triangleright$
Similarly to the above, from the inductive hypothesis for t_1 and t_2 we get:
$\Phi' \vdash t_1 \cdot \sigma : s$
$\Phi', t_1 \cdot \sigma \vdash \lceil t_2 \cdot \sigma \rceil_{|\Phi'|} : t' \cdot (\sigma, f_{|\Phi'|})$
From the inductive hypothesis for $\Pi(t_1).\, \lfloor t' \rfloor$ we get: $\Phi' \vdash (\Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma : s'$.
By the definition of $\cdot$ we get: $\Phi' \vdash \Pi(t_1 \cdot \sigma).(\lfloor t' \rfloor_{|\Phi+1|} \cdot \sigma) : s'$.
By the lemma A.22, we have that $(\lfloor t' \rfloor_{|\Phi+1|} \cdot \sigma) = \left\lfloor t' \cdot (\sigma, f_{|\Phi'|}) \right\rfloor_{|\Phi'|+1}$.
Thus we get $\Phi' \vdash \Pi(t_1 \cdot \sigma).\, \left\lfloor t' \cdot (\sigma, f_{|\Phi'|}) \right\rfloor_{|\Phi'|+1} : s'$.
We can now apply the same typing rule to get: $\Phi' \vdash \lambda(t_1 \cdot \sigma).(t_2 \cdot \sigma) : \Pi(t_1 \cdot \sigma).\, \left\lfloor t' \cdot (\sigma, f_{|\Phi'|}) \right\rfloor_{|\Phi'|+1}$.
We have $\Pi(t_1 \cdot \sigma).\, \left\lfloor t' \cdot (\sigma, f_{|\Phi'|}) \right\rfloor_{|\Phi'|+1} = \Pi(t_1 \cdot \sigma).((\lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma) = (\Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma$, thus this is the desired result.

Case $\dfrac{\Phi \vdash t_1 : \Pi(t).t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}$ $\triangleright$
By induction hypothesis for t_1 we get $\Phi' \vdash t_1 \cdot \sigma : \Pi(t \cdot \sigma).(t' \cdot \sigma)$.
By induction hypothesis for t_2 we get $\Phi' \vdash t_2 \cdot \sigma : t \cdot \sigma$.
By application of the same typing rule we get $\Phi' \vdash (t_1\ t_2) \cdot \sigma : \lceil t' \cdot \sigma \rceil_{|\Phi'|} \cdot (\mathsf{id}_{\Phi'}, t_2 \cdot \sigma)$.
We have that $\lceil t' \cdot \sigma \rceil_{|\Phi'|} \cdot (\mathsf{id}_{\Phi'}, t_2 \cdot \sigma) = (\lceil t' \rceil_{|\Phi|} \cdot (\sigma, f_{|\Phi'|})) \cdot (\mathsf{id}_{\Phi'},\, t_2 \cdot \sigma)$ due to lemma A.21
From lemma A.23 $(t \cdot \sigma) \cdot \sigma' = t \cdot (\sigma \cdot \sigma')$, we further have that the above is equal to $\lceil t' \rceil_{|\Phi|} \cdot ((\sigma, f_{|\Phi'|}) \cdot (\mathsf{id}_{\Phi'}, t_2 \cdot \sigma))$.
We will now prove that $((\sigma,\, f_{|\Phi'|}) \cdot (\mathsf{id}_{\Phi'},\, t_2 \cdot \sigma) = \sigma,\, (t_2 \cdot \sigma)$.
By definition we have $(\sigma, f_{|\Phi'|}) \cdot (\mathsf{id}_{\Phi'},\, t_2 \cdot \sigma) = (\sigma \cdot (\mathsf{id}_{\Phi'},\, t_2 \cdot \sigma)),\, (f_{|\Phi'|} \cdot (\mathsf{id}_{\Phi'},\, t_2 \cdot \sigma)) = (\sigma \cdot (\mathsf{id}_{\Phi'},\, t_2 \cdot \sigma)),\, t_2 \cdot \sigma$.
Due to lemma A.16, we have that $\sigma <^f |\Phi'|$. Thus from lemma A.19, we get that $\sigma \cdot (\mathsf{id}_{\Phi'},\, t_2) = \sigma \cdot \mathsf{id}_{\Phi'}$.
Last from lemma A.13 we get that $\sigma \cdot \mathsf{id}_{\Phi'} = \sigma$.
Thus we only need to show that $\lceil t' \rceil_{|\Phi|} \cdot (\sigma,\, (t_2 \cdot \sigma))$ is equal to $(\lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)) \cdot \sigma$.
As above, per lemma A.23, this is equal to $\lceil t' \rceil_{|\Phi|} \cdot ((\mathsf{id}_\Phi, t_2) \cdot \sigma)$.
From definition we have $((\mathsf{id}_\Phi, t_2) \cdot \sigma) = (\mathsf{id}_\Phi \cdot \sigma), (t_2 \cdot \sigma)$.
Furthermore, from lemma A.24 we get that $(\mathsf{id}_\Phi \cdot \sigma), (t_2 \cdot \sigma) = \sigma, (t_2 \cdot \sigma)$.
Thus we have the desired result.

Case (otherwise) $\triangleright$
Simple to prove based on the methods we have shown above.

Corollary A.27 *If* $\Phi' \vdash \sigma : \Phi$ *and* $\Phi'' \vdash \sigma' : \Phi'$ *then* $\Phi'' \vdash \sigma \cdot \sigma' : \Phi$.

Induction on the typing derivation for σ, with use of the substitution theorem A.26.

Lemma A.28 (Types are well-typed) *If* $\Phi \vdash t : t'$ *then either* $t' = \mathsf{Type}'$ *or* $\Phi \vdash t' : s$.

By structural induction on the typing derivation for t.

Case $\dfrac{c : t \in \Sigma}{\Phi \vdash_\Sigma c : t}$ $\triangleright$ Trivial by inversion of the well-formedness of Σ.

Case $\dfrac{\Phi.i = t}{\Phi \vdash f_i : t}$ $\triangleright$ Trivial by inversion of the well-formedness of Φ.

Case $\dfrac{(s,s') \in \mathcal{A}}{\Phi \vdash s : s'}$ $\triangleright$ By splitting cases for (s,s') and application of the same typing rule.

Case $\dfrac{\Phi \vdash t_1 : s \qquad \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s,s',s'') \in \mathcal{R}}{\Phi \vdash \Pi(t_1).t_2 : s''}$ $\triangleright$ By splitting cases for (s,s',s'') and use of sort typing rule.

Case $\dfrac{\Phi \vdash t_1 : \Pi(t).t' \qquad \Phi \vdash t_2 : t}{\Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}$ $\triangleright$
By induction hypothesis we get that $\Phi \vdash \Pi(t).t' : s$. By inversion of this judgement, we get that $\Phi,\, t \vdash \lceil t' \rceil : s'$.
Furthermore we have by lemma A.15 that $\Phi \vdash \mathsf{id}_{|\Phi|} : \Phi$, and using the typing for t_2 and lemma A.12, we get that $\Phi \vdash \mathsf{id}_{|\Phi|},\, t_2 : (\Phi,\, t)$.
Thus by application of the substitution lemma A.26 for $\lceil t' \rceil$ we get the desired result.

Case (otherwise) $\triangleright$ Simple to prove based on the methods we have shown above.

Lemma A.29 (Weakening) *If $\Phi \vdash t : t'$, then $\Phi, t_1, t_2, \cdots, t_n \vdash t : t'$.*

Using lemma A.15 we have that $\Phi, t_1, t_2, \cdots, t_n \vdash \mathsf{id}_\Phi : \Phi$.
Using the substitution lemma A.26 we get that $\Phi, t_1, t_2, \cdots, t_n \vdash t \cdot \mathsf{id}_\Phi : t' \cdot \mathsf{id}_\Phi$.
From lemma A.18 and A.14, we get that $t \cdot \mathsf{id}_\Phi = t$.
From lemma A.28 we further get $\Phi \vdash t' : s$ and applying the same lemmas as for t we get $t' \cdot \mathsf{id}_\Phi = t'$.

B. Extension with metavariables and polymorphic contexts

B.1 Extending with metavariables

First, we extend the previous definition of terms to account for metavariables.

Definition B.1 (Syntax of the language) *The syntax of the logic language is extended below. We furthermore add new syntactic classes for modal terms and environments of metavariables.*

$$
\begin{aligned}
t &::= \cdots \mid X_i/\sigma \\
T &::= [\Phi]t \\
\mathcal{M} &::= \bullet \mid \mathcal{M},\, T
\end{aligned}
$$

Now we gather all the places from the above section where something was defined through induction on terms, and redefine/extend them here. Things that are identical are noted.

Definition B.2 (Context length and access) *Identical to A.2. We furthermore define metavariables environment length and access here.*

$\boxed{|\mathcal{M}|}$

$$\begin{array}{lcl} |\bullet| & = & 0 \\ |\mathcal{M},\, T| & = & |\mathcal{M}| + 1 \end{array}$$

$\boxed{\mathcal{M}.i}$

$$\begin{array}{lcl} (\mathcal{M},\, T).|\mathcal{M}| & = & T \\ (\mathcal{M},\, T).i & = & \mathcal{M}.i \end{array}$$

Definition B.3 (Substitution length) *Identical to A.3.*

Definition B.4 (Substitution access) *Identical to A.4.*

Definition B.5 (Substitution application) *This is the extension of definition A.5. We lift it to modal terms.*

$\boxed{t \cdot \sigma}$

$$(X_i/\sigma') \cdot \sigma \quad = \quad X_i/(\sigma' \cdot \sigma)$$

$\boxed{T \cdot \sigma}$

$$([\Phi]t) \cdot \sigma \quad = \quad t \cdot \sigma$$

Definition B.6 (Identity substitution) *Identical to A.6.*

Definition B.7 (Variable limits for terms and substitutions) *This is the extension of definition A.7 and definition A.8 (who are now mutually dependent). The definition for substitutions is identical.*

$\boxed{t <^f n}$

$$X_i/\sigma <^f n \quad \Leftarrow \quad \sigma <^f n$$

$\boxed{t <^b n}$

$$X_i/\sigma <^b n \quad \Leftarrow \quad \sigma <^b n$$

Definition B.8 (Freshening) *This is the extension of definition A.9. Furthermore we need to lift the freshening operation to substitutions.*

$\boxed{\lceil t \rceil_m^n}$

$$\lceil X_i/\sigma \rceil_m^n \quad = \quad X_i/(\lceil \sigma_m^n \rceil)$$

$\boxed{\lceil \sigma \rceil_m^n}$

$$\begin{array}{lcl} \lceil \bullet \rceil_m^n & = & \bullet \\ \lceil \sigma,\, t \rceil_m^n & = & (\lceil \sigma \rceil_m^n),\, \lceil t \rceil_m^n \end{array}$$

Definition B.9 (Binding) *This is the extension of definition A.10. As above, we need to lift binding to substitutions.*

$\boxed{\lfloor t \rfloor^n_m}$

$$\lfloor X_i/\sigma \rfloor^n_m \;=\; X_i/(\lfloor \sigma^n_m \rfloor)$$

$\boxed{\lfloor \sigma \rfloor^n_m}$

$$\begin{array}{lcl} \lfloor \bullet \rfloor^n_m & = & \bullet \\ \lfloor \sigma,\, t \rfloor^n_m & = & (\lfloor \sigma \rfloor^n_m),\ \lfloor t \rfloor^n_m \end{array}$$

Definition B.10 (Typing judgements) *The typing judgements defined in A.11 are adjusted as follows.*
First, the judgement $\Phi \vdash t : t'$ is replaced by the judgement $\mathcal{M};\ \Phi \vdash t : t'$ and the existing rules are adjusted as needed. Also we include a new rule shown below.
Second, the judgement $\vdash \Phi$ wf is replaced by the judgement $\mathcal{M} \vdash \Phi$ wf.
Third, the judgement $\mathcal{M};\ \Phi \vdash \sigma : \Phi'$ replaces the original judgement for substitutions.
The $\vdash \Sigma$ wf judgement stays as is, with the adjustment shown below.
Last, we introduce a new judgement $\vdash \mathcal{M}$ wf for meta-environments and a judgement $\mathcal{M} \vdash T : T'$ for modal terms.

$\boxed{\vdash \Sigma \text{ wf}}$

$$\frac{}{\vdash \Sigma \text{ wf}} \qquad\qquad \frac{\vdash \Sigma \text{ wf} \qquad \bullet;\bullet \vdash_\Sigma t : s \qquad (c :) \notin \Sigma}{\vdash (\Sigma,\ c : t) \text{ wf}}$$

$\boxed{\mathcal{M};\ \Phi \vdash t : t'}$

$$\frac{\mathcal{M}.i = T \qquad T = [\Phi']t' \qquad \mathcal{M};\ \Phi \vdash \sigma : \Phi'}{\mathcal{M};\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}$$

$\boxed{\vdash \mathcal{M} \text{ wf}}$

$$\vdash \bullet \text{ wf} \qquad\qquad \frac{\vdash \mathcal{M} \text{ wf} \qquad \mathcal{M} \vdash [\Phi]t : [\Phi]s}{\vdash (\mathcal{M},\ [\Phi]t) \text{ wf}}$$

$\boxed{\mathcal{M} \vdash T : T'}$

$$\frac{\mathcal{M};\ \Phi \vdash t : t'}{\mathcal{M} \vdash [\Phi]t : [\Phi]t'}$$

We can now proceed to adjust the proofs from above in order to handle the additional cases of the extension.

Lemma B.11 (Extension of lemmas A.12 and A.13) *1. If $t <^f m$ and $|\Phi| = m$ then $t \cdot id_\Phi = t$.*
2. If $\sigma <^f m$ and $|\Phi| = m$ then $\sigma \cdot id_\Phi = \sigma$.

The two lemmas become mutually dependent. For the first part, we proceed as previously by induction on t, and the only additional case we need to take into account is for the extension[1]:
We have that $(X_i/\sigma) \cdot \mathsf{id}_m = X_i/(\sigma \cdot \mathsf{id}_m)$. Using the second part, we have that $X_i/(\sigma \cdot \mathsf{id}_m) = X_i/\sigma$. The second part is proved as previously.

[1] We will not note this any more below; all the proofs mimic the inductive structure of the base proofs

Lemma B.12 (Extension of lemmas A.14 and A.16) *1. If* $\mathcal{M};\ \Phi \vdash t : t'$ *then* $t <^f |\Phi|$ *and* $t <^b 0$.

2. If $\mathcal{M};\ \Phi \vdash \sigma : \Phi'$ *then* $\sigma <^f |\Phi|$, $\sigma <^b 0$ *and* $|\sigma| = |\Phi'|$.

Again the two lemmas become mutually dependent when they weren't before. For the first one, we have that $\mathcal{M};\ \Phi \vdash X_i/\sigma : t'$; using the second part, we have that $\sigma <^f |\Phi|$ and $\sigma <^b 0$. By definition we thus have $X_i/\sigma <^f |\Phi|$ and $X_i/\sigma <^b 0$. The second part is proved as previously.

Lemma B.13 (Extension of lemma A.15) *If* $\mathcal{M} \vdash \Phi$ *wf then for any* Φ' *and* $t_{1\cdots n}$ *such that* $\Phi' = \Phi,\ t_1,\ t_2, \cdots, t_n$ *and* $\mathcal{M} \vdash \Phi'$ *wf, we have that* $\mathcal{M};\ \Phi' \vdash id_\Phi : \Phi$.

Identical as before.

Lemma B.14 (Extension of lemma A.17) *If* $\mathcal{M} \vdash \Phi$ *wf and* $|\Phi| = n$ *then for all* $i < n$, $\Phi.i <^f i$.

Identical as before.

Lemma B.15 (Extension of lemmas A.18 and A.19) *1. If* $t <^f m$, $|\sigma| = m$ *and* $t \cdot \sigma = t'$ *then* $t \cdot (\sigma, t_1, t_2, \cdots, t_n) = t'$.

2. If $\sigma <^f m$, $|\sigma'| = m$ *and* $\sigma \cdot \sigma' = \sigma_r$ *then* $\sigma \cdot (\sigma', t_1, t_2, \cdots, t_n) = \sigma_r$.

For the first part, taking $t = X_i/\sigma'$, we have that $X/\sigma' <^f m$ and thus $\sigma' <^f m$.
Furthermore we have $(X_i/\sigma') \cdot \sigma = X_i/(\sigma' \cdot \sigma) = X_i/\sigma_r$, assuming $\sigma_r = \sigma' \cdot \sigma$.
Using the second lemma we have that $\sigma' \cdot (\sigma, t_1, t_2, \cdots, t_n) = \sigma_r$.
Thus we also have that $(X_i/\sigma') \cdot (\sigma, t_1, t_2, \cdots, t_n) = X_i/(\sigma' \cdot (\sigma, t_1, t_2, \cdots, t_n)) = X_i/\sigma_r$.
For the second part, the proof proceeds as previously.

Lemma B.16 (Extension of lemma A.20) *If* $\mathcal{M} \vdash \Phi$ *wf,* $\Phi.i = t$ *and* $\mathcal{M};\ \Phi' \vdash \sigma : \Phi$, *then* $\mathcal{M};\ \Phi' \vdash \sigma.i : t \cdot \sigma$.

Identical as before.

Lemma B.17 (Extension of lemma A.21 and new lemma for substitutions) *1. If* $t <^f m$, $t <^b n+1$, $\sigma <^f m'$ *and* $|\sigma| = m$ *then* $\lceil t \cdot \sigma \rceil^n_{m'} = \lceil t \rceil^n_m \cdot (\sigma, f_{m'})$.

2. If $\sigma' <^f m$, $\sigma' <^b n+1$, $\sigma <^f m'$ *and* $|\sigma| = m$ *then* $\lceil \sigma' \cdot \sigma \rceil^n_{m'} = \lceil \sigma' \rceil^n_m \cdot (\sigma, f_{m'})$.

The second part of this lemma is a new lemma; it corresponds to the lifting of the first part to substitutions.
For the first part, we have: $\lceil (X_i/\sigma') \cdot \sigma \rceil^n_{m'} = \lceil X_i/(\sigma' \cdot \sigma) \rceil^n_{m'} = X_i/\lceil \sigma' \cdot \sigma \rceil^n_{m'}$.
Using the second part, we have that this is equal to $X_i/(\lceil \sigma' \rceil^n_m \cdot (\sigma,\ f_{m'}))$.
Furthermore, this is equal to $(X_i/\lceil \sigma' \rceil^n_m) \cdot (\sigma,\ f_{m'})$.
Last, this is equal to $(\lceil X_i/\sigma' \rceil^n_m) \cdot (\sigma,\ f_{m'})$, which is the desired.

For the second part, we proceed by induction on σ'.
If $\sigma' = \bullet$, the result is trivial.
If $\sigma' = \sigma'', t$ then $\lceil (\sigma'',\ t) \cdot \sigma \rceil^n_{m'} = \lceil (\sigma'' \cdot \sigma),\ t \cdot \sigma \rceil^n_{m'} = \lceil \sigma'' \cdot \sigma \rceil^n_{m'},\ \lceil t \cdot \sigma \rceil^n_{m'}$.
Using the induction hypothesis and the first part, we have that this is equal to $\lceil \sigma'' \rceil^n_m \cdot (\sigma,\ f_{m'}),\ \lceil t \rceil^n_m \cdot (\sigma,\ f_{m'}) = \lceil \sigma'',\ t \rceil^n_m \cdot (\sigma,\ f_{m'})$, which is the desired.

Lemma B.18 (Extension of lemma A.22 and new lemma for substitutions) *1. If* $t <^f m+1$, $t <^b n$, $\sigma <^f m'$ *and* $|\sigma| = m$ *then* $\lfloor t \cdot (\sigma, f_{m'}) \rfloor^n_{m'+1} = \lfloor t \rfloor^n_{m+1} \cdot \sigma$.

2. If $\sigma' <^f m+1$, $\sigma' <^b n$, $\sigma <^f m'$ *and* $|\sigma| = m$ *then* $\lfloor \sigma' \cdot (\sigma, f_{m'}) \rfloor^n_{m'+1} = \lfloor \sigma' \rfloor^n_{m+1} \cdot \sigma$.

This proof is entirely similar to the above for both parts.

Lemma B.19 (Extension of lemma A.23 and new lemma for substitutions) *1. If $t <^f m$, $|\sigma| = m$, $\sigma <^f m'$ and $|\sigma'| = m'$ then $(t \cdot \sigma) \cdot \sigma' = t \cdot (\sigma \cdot \sigma')$.*

2. If $\sigma_1 <^f m$, $|\sigma| = m$, $\sigma <^f m'$ and $|\sigma'| = m'$ then $(\sigma_1 \cdot \sigma) \cdot \sigma' = \sigma_1 \cdot (\sigma \cdot \sigma')$.

Entirely similar to the above.

Lemma B.20 (Extension of lemma A.24) *If $|\sigma| = m$ and $|\Phi| = m$ then $id_\Phi \cdot \sigma = \sigma$.*

Identical as before.

Lemma B.21 (Extension of lemma A.25) *1. If $\lceil t \rceil_m^n = \lceil t' \rceil_m^n$ then $t = t'$.*

2. If $\lceil \sigma \rceil_m^n = \lceil \sigma' \rceil_m^n$ then $\sigma = \sigma'$.

Part 1 is identical as before, with the additional case $t = X_i/\sigma$ and $t' = X_i/\sigma'$ handled using the second part. Part 2 is proved by induction on the structure of σ.

Theorem B.22 (Extension of main substitution theorem A.26 and corollary A.27) *1. If $\mathcal{M};\ \Phi \vdash t : t'$ and $\mathcal{M};\ \Phi' \vdash \sigma : \Phi$ then $\mathcal{M};\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma$.*

2. If $\mathcal{M};\ \Phi' \vdash \sigma : \Phi$ and $\mathcal{M};\ \Phi'' \vdash \sigma' : \Phi'$ then $\mathcal{M};\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi$.

3. If $\mathcal{M} \vdash [\Phi']t : [\Phi']t'$ and $\mathcal{M};\ \Phi \vdash \sigma : \Phi'$ then $\mathcal{M} \vdash [\Phi]t \cdot \sigma : [\Phi]t' \cdot \sigma$.

For the first part we have, when $t = X_i/\sigma_0$:
From $\mathcal{M};\ \Phi \vdash X_i/\sigma_0 : t'$ we get that $\mathcal{M}.i = [\Phi_0]t_0$, $\mathcal{M};\ \Phi \vdash \sigma_0 : \Phi_0$ and $t' = t_0 \cdot \sigma_0$.
Applying the second part of the lemma for $\sigma = \sigma_0$ and $\sigma' = \sigma$ we get that $\mathcal{M};\ \Phi' \vdash \sigma_0 \cdot \sigma' : \Phi_0$.
Thus applying the same typing rule for $t = X_i/(\sigma_0 \cdot \sigma)$ we get that $\mathcal{M};\ \Phi' \vdash X_i/(\sigma_0 \cdot \sigma') : t_0 \cdot (\sigma_0 \cdot \sigma')$.
Taking into account the definition of $\cdot$ and also lemma B.19, we have that this is the desired result.

For the second part, the proof is identical to the proof done earlier.
For the third part, by typing inversion for $[\Phi']t$ we get that $\mathcal{M};\ \Phi' \vdash t : t'$.
Using the first part we get that $\mathcal{M};\ \Phi \vdash t \cdot \sigma : t' \cdot \sigma$.
Using the typing rule for modal terms we get $\mathcal{M} \vdash [\Phi]t \cdot \sigma : [\Phi]t' \cdot \sigma$.

Lemma B.23 (Meta-variables context weakening) *1. If $\mathcal{M};\Phi \vdash t : t'$ then $\mathcal{M}, T_1, \cdots, T_n;\ \Phi \vdash t : t'$.*

2. If $\mathcal{M};\Phi \vdash \sigma : \Phi'$ then $\mathcal{M}, T_1, \cdots, T_n;\ \Phi \vdash \sigma : \Phi'$.

3. If $\mathcal{M} \vdash \Phi$ wf then $\mathcal{M}, T_1, \cdots, T_n \vdash \Phi$ wf.

4. If $\mathcal{M} \vdash T : T'$ then $\mathcal{M}, T_1, \cdots, T_n \vdash T : T'$.

All are trivial by induction on the typing derivations.

Lemma B.24 (Extension of lemma A.28) *If $\mathcal{M};\ \Phi \vdash t : t'$ then either $t' = \mathsf{Type}'$ or $\mathcal{M};\ \Phi \vdash t' : s$.*

When $t = X_i/\sigma$, by inversion of typing we get $\mathcal{M}.i = [\Phi']t''$, $\mathcal{M};\ \Phi \vdash \sigma : \Phi'$ and $t' = t'' \cdot \sigma$.
By inversion of well-formedness for $\mathcal{M}$ and lemma 4, we get that $\mathcal{M} \vdash \mathcal{M}.i : [\Phi']s$.
Furthermore by inversion of that we get $\mathcal{M};\ \Phi' \vdash t'' : s$.
By application of the substitution lemma B.22 for t'' and σ we get $\mathcal{M};\ \Phi \vdash t'' \cdot \sigma : s$, which is the desired result.

Lemma B.25 (Extension of the lemma A.29 and new lemma for substitutions) *1. If* $\mathcal{M};\ \Phi \vdash t : t'$ *then* $\mathcal{M};\ \Phi, t_1, t_2, \cdots, t_n \vdash t : t'$.

2. If $\mathcal{M};\ \Phi \vdash \sigma : \Phi'$ *then* $\mathcal{M};\ \Phi, t_1, t_2, \cdots, t_n \vdash \sigma : \Phi'$.

For the first part, proceed identically as before.
For the second part, the proof is entirely similar to the first part (construct and prove well-typedness of identity substitution, and then allude to substitution theorem).

Now we know that everything that all the theorems we had proved for the non-extended version still hold. We can now prove a new meta-substitution theorem. Before doing that we need some new definitions.

Definition B.26 (Substitutions of meta-variables) *The syntax of substitutions of meta-variables is defined as follows.*

$$\sigma_{\mathcal{M}} ::= \bullet \mid \sigma_{\mathcal{M}},\ T$$

Definition B.27 (Meta-substitution length and access) *We define the length of meta-substitutions and accessing the i-th element as follows.*

$\boxed{|\sigma_{\mathcal{M}}|}$

$$\begin{array}{lcl} |\bullet| & = & 0 \\ |\sigma_{\mathcal{M}},\ T| & = & |\sigma_{\mathcal{M}}| + 1 \end{array}$$

$\boxed{\sigma_{\mathcal{M}}.i}$

$$\begin{array}{lcl} (\sigma_{\mathcal{M}},\ T).|\sigma_{\mathcal{M}}| & = & T \\ (\sigma_{\mathcal{M}},\ T).i & = & \sigma_{\mathcal{M}}.i \end{array}$$

Definition B.28 (Meta-substitution application) *The application of meta-substitutions is defined as follows. We mark the interesting cases with a star.*

$\boxed{t \cdot \sigma_{\mathcal{M}}}$

$$\begin{array}{lcl} s \cdot \sigma_{\mathcal{M}} & = & s \\ c \cdot \sigma_{\mathcal{M}} & = & c \\ f_i \cdot \sigma_{\mathcal{M}} & = & f_i \\ b_i \cdot \sigma_{\mathcal{M}} & = & b_i \\ (\lambda(t_1).t_2) \cdot \sigma_{\mathcal{M}} & = & \lambda(t_1 \cdot \sigma_{\mathcal{M}}).(t_2 \cdot \sigma_{\mathcal{M}}) \\ (t_1\ t_2) \cdot \sigma_{\mathcal{M}} & = & (t_1 \cdot \sigma_{\mathcal{M}})\ (t_2 \cdot \sigma_{\mathcal{M}}) \\ (\Pi(t_1).t_2) \cdot \sigma_{\mathcal{M}} & = & \Pi(t_1 \cdot \sigma_{\mathcal{M}}).(t_2 \cdot \sigma_{\mathcal{M}}) \end{array}$$

$\boxed{t \cdot \sigma_{\mathcal{M}} \text{ (continued)}}$

$$
\begin{array}{lll}
(t_1 = t_2) \cdot \sigma_{\mathcal{M}} & = & (t_1 \cdot \sigma_{\mathcal{M}}) = (t_2 \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{conv}\ t_1\ t_2) \cdot \sigma_{\mathcal{M}} & = & \mathsf{conv}\ (t_1 \cdot \sigma_{\mathcal{M}})\ (t_2 \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{refl}\ t) \cdot \sigma_{\mathcal{M}} & = & \mathsf{refl}\ (t \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{symm}\ t) \cdot \sigma_{\mathcal{M}} & = & \mathsf{symm}\ (t \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{trans}\ t_1\ t_2) \cdot \sigma_{\mathcal{M}} & = & \mathsf{trans}\ (t_1 \cdot \sigma_{\mathcal{M}})\ (t_2 \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{congapp}\ t_1\ t_2) \cdot \sigma_{\mathcal{M}} & = & \mathsf{congapp}\ (t_1 \cdot \sigma_{\mathcal{M}})\ (t_2 \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{congimpl}\ t_1\ t_2) \cdot \sigma_{\mathcal{M}} & = & \mathsf{congimpl}\ (t_1 \cdot \sigma_{\mathcal{M}})\ (t_2 \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{conglam}\ t) \cdot \sigma_{\mathcal{M}} & = & \mathsf{conglam}\ (t \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{congpi}\ t) \cdot \sigma_{\mathcal{M}} & = & \mathsf{congpi}\ (t \cdot \sigma_{\mathcal{M}}) \\
(\mathsf{beta}\ t_1\ t_2) \cdot \sigma_{\mathcal{M}} & = & \mathsf{beta}\ (t_1 \cdot \sigma_{\mathcal{M}})\ (t_2 \cdot \sigma_{\mathcal{M}}) \\
* \quad (X_i/\sigma) \cdot \sigma_{\mathcal{M}} & = & (\sigma_{\mathcal{M}}.i) \cdot (\sigma \cdot \sigma_{\mathcal{M}})
\end{array}
$$

$\boxed{\sigma \cdot \sigma_{\mathcal{M}}}$

$$
\begin{array}{lll}
\bullet \cdot \sigma_{\mathcal{M}} & = & \bullet \\
(\sigma,\ t) \cdot \sigma_{\mathcal{M}} & = & \sigma \cdot \sigma_{\mathcal{M}},\ t \cdot \sigma_{\mathcal{M}}
\end{array}
$$

$\boxed{\Phi \cdot \sigma_{\mathcal{M}}}$

$$
\begin{array}{lll}
\bullet \cdot \sigma_{\mathcal{M}} & = & \bullet \\
(\Phi,\ t) \cdot \sigma_{\mathcal{M}} & = & \Phi \cdot \sigma_{\mathcal{M}},\ t \cdot \sigma_{\mathcal{M}}
\end{array}
$$

$\boxed{T \cdot \sigma_{\mathcal{M}}}$

$$
* \quad ([\Phi]\, t) \cdot \sigma_{\mathcal{M}} \quad = \quad [\Phi \cdot \sigma_{\mathcal{M}}]\, (t \cdot \sigma_{\mathcal{M}})
$$

Definition B.29 (Meta-substitution typing) *The typing judgement for meta-substitutions is as follows.*

$\boxed{\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'}$

$$
\frac{}{\mathcal{M} \vdash \bullet : \bullet}
\qquad\qquad
\frac{\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}' \qquad \mathcal{M} \vdash T : T' \cdot \sigma_{\mathcal{M}}}{\mathcal{M} \vdash (\sigma_{\mathcal{M}},\ T) : (\mathcal{M}',\ T')}
$$

We proceed to prove the meta-substitution theorem.

The lemmas that we need are the following:

Lemma B.30 (Limits for elements of metasubstitutions) *If $\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'$ and $\sigma_{\mathcal{M}}.i = [\Phi]\, t$ then $t <^f |\Phi|$ and $t <^b 0$.*

By repeated inversion of typing for $\sigma_{\mathcal{M}}$ we get that $\mathcal{M}' \vdash \sigma_{\mathcal{M}}.i : T'$ for some $\mathcal{M}'$ and T'. By inversion we get that $\mathcal{M}';\ \Phi \vdash t : t'$. By use of lemma 2 we get the desired.

Lemma B.31 (Freshen on closed term) *If $t <^b n$ then $\lceil t \cdot \sigma \rceil^n_m = t \cdot \lceil \sigma \rceil^n_m$.*

Easy by induction on t.

Lemma B.32 (Interaction of freshen and metasubstitution application) *1. If* $\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'$ *then* $\lceil t \rceil^n_m \cdot \sigma_{\mathcal{M}} = \lceil t \cdot \sigma_{\mathcal{M}} \rceil^n_m$

2. If $\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'$ *then* $\lceil \sigma \rceil^n_m \cdot \sigma_{\mathcal{M}} = \lceil \sigma \cdot \sigma_{\mathcal{M}} \rceil^n_m$

The first part is proved by induction on t. The interesting case is the metavariables case, where we have the following.
$\lceil X_i/\sigma \rceil^n_m \cdot \sigma_{\mathcal{M}} = (X_i/\lceil \sigma \rceil^n_m) \cdot \sigma_{\mathcal{M}} = \sigma_{\mathcal{M}}.i \cdot (\lceil \sigma \rceil^n_m \cdot \sigma_{\mathcal{M}}) = \sigma_{\mathcal{M}}.i \cdot \lceil \sigma \cdot \sigma_{\mathcal{M}} \rceil^n_m$ based on the second part.
Now $\sigma_{\mathcal{M}}.i = [\Phi]t$ and the above is further equal to: $t \cdot \lceil \sigma \cdot \sigma_{\mathcal{M}} \rceil^n_m$. The right-hand side is rewritten as follows:
$\lceil X_i/\sigma \cdot \sigma_{\mathcal{M}} \rceil^n_m = \lceil \sigma_{\mathcal{M}}.i \cdot (\sigma \cdot \sigma_{\mathcal{M}}) \rceil^n_m = \lceil t \cdot (\sigma \cdot \sigma_{\mathcal{M}}) \rceil^n_m = t \cdot \lceil \sigma \cdot \sigma_{\mathcal{M}} \rceil^n_m$ using lemma B.31 and also B.30.
The second part is proved trivially using induction.

Lemma B.33 (Bind on closed term) *If* $t <^b n$ *then* $\lfloor t \cdot \sigma \rfloor^n_m = t \cdot \lfloor \sigma \rfloor^n_m$.

Easy by induction on t.

Lemma B.34 (Interaction of bind and metasubstitution application) *1. If* $\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'$ *then* $\lfloor t \rfloor^n_m \cdot \sigma_{\mathcal{M}} = \lfloor t \cdot \sigma_{\mathcal{M}} \rfloor^n_m$

2. If $\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'$ *then* $\lfloor \sigma \rfloor^n_m \cdot \sigma_{\mathcal{M}} = \lfloor \sigma \cdot \sigma_{\mathcal{M}} \rfloor^n_m$

Similar to the equivalent lemma for freshen.

Lemma B.35 (Interaction of substitution application and metasubstitution application) *1.* $(t \cdot \sigma) \cdot \sigma_{\mathcal{M}} = (t \cdot \sigma_{\mathcal{M}}) \cdot (\sigma \cdot \sigma_{\mathcal{M}})$

2. $(\sigma \cdot \sigma') \cdot \sigma_{\mathcal{M}} = (\sigma \cdot \sigma_{\mathcal{M}}) \cdot (\sigma' \cdot \sigma_{\mathcal{M}})$

In the first part, we perform induction on t. The interesting case is the metavariables case. We have:
$((X_i/\sigma') \cdot \sigma) \cdot \sigma_{\mathcal{M}} = (X_i/(\sigma' \cdot \sigma)) \cdot \sigma_{\mathcal{M}} = \sigma_{\mathcal{M}}.i \cdot ((\sigma' \cdot \sigma) \cdot \sigma_{\mathcal{M}})$.
From the second part, this is equal to: $\sigma_{\mathcal{M}}.i \cdot ((\sigma' \cdot \sigma_{\mathcal{M}}) \cdot (\sigma \cdot \sigma_{\mathcal{M}}))$.
There exists a t such that $\sigma_{\mathcal{M}}.i = [\Phi]t$ and thus the above is further equal to:
$t \cdot ((\sigma' \cdot \sigma_{\mathcal{M}}) \cdot (\sigma \cdot \sigma_{\mathcal{M}})) = (t \cdot (\sigma' \cdot \sigma_{\mathcal{M}})) \cdot (\sigma \cdot \sigma_{\mathcal{M}})$ based on lemma B.19.
The right-hand side is written as: $((X_i/\sigma') \cdot \sigma_{\mathcal{M}}) \cdot (\sigma \cdot \sigma_{\mathcal{M}}) = (t \cdot (\sigma' \cdot \sigma_{\mathcal{M}})) \cdot (\sigma \cdot \sigma_{\mathcal{M}})$. Thus the desired.
The second part is trivially proved by induction and use of the first part.

Lemma B.36 (Application of metasubstitution to identity substitution) $id_\Phi \cdot \sigma_{\mathcal{M}} = id_{\Phi \cdot \sigma_{\mathcal{M}}}$

Trivial by induction on Φ.

Lemma B.37 (Redundant elements in metasubstitutions) *1. If* $\mathcal{M}; \Phi \vdash t : t'$ *and* $|\sigma_{\mathcal{M}}| = |\mathcal{M}|$ *then* $t \cdot (\sigma_{\mathcal{M}}, T_1, T_2, \cdots, T_n) = t \cdot \sigma_{\mathcal{M}}$.

2. If $\mathcal{M}; \Phi \vdash \sigma : \Phi'$ *and* $|\sigma_{\mathcal{M}}| = |\mathcal{M}|$ *then* $\sigma \cdot (\sigma_{\mathcal{M}}, T_1, T_2, \cdots, T_n) = \sigma \cdot \sigma_{\mathcal{M}}$.

3. If $\mathcal{M} \vdash \Phi$ *wf and* $|\sigma_{\mathcal{M}}| = |\mathcal{M}|$ *then* $\Phi \cdot (\sigma_{\mathcal{M}}, T_1, T_2, \cdots, T_n) = \Phi \cdot \sigma_{\mathcal{M}}$.

4. If $\mathcal{M} \vdash T : T'$ *and* $|\sigma_{\mathcal{M}}| = |\mathcal{M}|$ *then* $T \cdot (\sigma_{\mathcal{M}}, T_1, T_2, \cdots, T_n) = T \cdot \sigma_{\mathcal{M}}$.

By induction on the typing derivations.

Lemma B.38 (Type of i-th metasubstitution element) *If* $\vdash \mathcal{M}$ *wf and* $\mathcal{M} \vdash \sigma_{\mathcal{M}} : \mathcal{M}'$ *then* $\mathcal{M} \vdash \sigma_{\mathcal{M}}.i : (\mathcal{M}'.i) \cdot \sigma_{\mathcal{M}}$.

By induction and use of lemma B.37; furthermore using inversion of the well-formedness relation for $\mathcal{M}$. Similar to lemma A.20.

Theorem B.39 (Substitution over metavariables) *1. If* $\mathcal{M};\ \Phi \vdash t : t'$ *and* $\mathcal{M}' \vdash \sigma_{\mathcal{M}} : \mathcal{M}$ *then* $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} : t \cdot \sigma_{\mathcal{M}} : t' \cdot \sigma_{\mathcal{M}}$.

2. If $\mathcal{M};\ \Phi \vdash \sigma : \Phi'$ *and* $\mathcal{M}' \vdash \sigma_{\mathcal{M}} : \mathcal{M}$ *then* $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} \vdash \sigma \cdot \sigma_{\mathcal{M}} : \Phi' \cdot \sigma_{\mathcal{M}}$.

3. If $\mathcal{M} \vdash \Phi$ *wf and* $\mathcal{M}' \vdash \sigma_{\mathcal{M}} : \mathcal{M}$ *then* $\mathcal{M}' \vdash \Phi \cdot \sigma_{\mathcal{M}}$ *wf.*

4. If $\mathcal{M} \vdash T : T'$ *and* $\mathcal{M}' \vdash \sigma_{\mathcal{M}} : \mathcal{M}$ *then* $\mathcal{M}' \vdash T \cdot \sigma_{\mathcal{M}} : T' \cdot \sigma_{\mathcal{M}}$.

Part 1 Proceed by structural induction on the typing of t.

Case $\dfrac{c : t \in \Sigma}{\mathcal{M};\ \Phi \vdash_{\Sigma} c : t}\ \triangleright$
From inversion of the well-formedness of Σ we have that $\bullet;\bullet \vdash t : s$.
From lemma B.37 we have that $t \cdot \sigma_{\mathcal{M}} = t$.
So the result follows from application of the same typing rule for $\Phi \cdot \sigma_{\mathcal{M}}$.

Case $\dfrac{\Phi.i = t}{\mathcal{M};\ \Phi \vdash f_i : t}\ \triangleright$
We have $t \cdot \sigma_{\mathcal{M}} = (\Phi \cdot \sigma_{\mathcal{M}}).i$, so using the same typing rule we get $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} \vdash f_i : t \cdot \sigma_{\mathcal{M}}$.

Case $\dfrac{(s,s') \in \mathcal{A}}{\mathcal{M};\ \Phi \vdash s : s'}\ \triangleright$
Trivial by application of the same rule and the definition of $\cdot$.

Case $\dfrac{\mathcal{M};\ \Phi \vdash t_1 : s \qquad \mathcal{M};\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s,s',s'') \in \mathcal{R}}{\mathcal{M};\ \Phi \vdash \Pi(t_1).t_2 : s''}\ \triangleright$
By induction hypothesis for t_1 we get: $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} \vdash t_1 \cdot \sigma_{\mathcal{M}} : s$.
By induction hypothesis for $\Phi, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s'$ we get:
$\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}}, t_1 \cdot \sigma_{\mathcal{M}} \vdash \lceil t_2 \rceil_{|\Phi|} \cdot \sigma_{\mathcal{M}} : s' \cdot \sigma_{\mathcal{M}}$.
We have $s' = s' \cdot \sigma_{\mathcal{M}}$ trivially.
Also by the lemma B.32, $\lceil t_2 \rceil_{|\Phi|} \cdot \sigma_{\mathcal{M}} = \lceil t_2 \cdot \sigma_{\mathcal{M}} \rceil_{|\Phi|}$.
Thus by application of the same typing rule we get $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} \vdash \Pi(t_1 \cdot \sigma_{\mathcal{M}}).(t_2 \cdot \sigma_{\mathcal{M}}) : s''$ which is the desired.

Case $\dfrac{\mathcal{M};\ \Phi \vdash t_1 : s \qquad \mathcal{M};\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \mathcal{M};\ \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1} : s'}{\mathcal{M};\ \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}}\ \triangleright$
Similarly to the above, from the inductive hypothesis for t_1 and t_2 (and use of lemma B.32) we get:
$\sigma_{\mathcal{M}};\ \Phi \vdash t_1 \cdot \sigma_{\mathcal{M}} : s$
$\sigma_{\mathcal{M}};\ \Phi \cdot \sigma_{\mathcal{M}}, t_1 \cdot \sigma_{\mathcal{M}} \vdash \lceil t_2 \cdot \sigma_{\mathcal{M}} \rceil_{|\Phi|} : t' \cdot \sigma_{\mathcal{M}}$
From the inductive hypothesis for $\Pi(t_1).\ \lfloor t' \rfloor$ we get: $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} \vdash (\Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma_{\mathcal{M}} : s'$.
By the definition of $\cdot$ we get: $\mathcal{M}';\ \Phi \cdot \sigma_{\mathcal{M}} \vdash \Pi(t_1 \cdot \sigma_{\mathcal{M}}).(\lfloor t' \rfloor_{|\Phi+1|} \cdot \sigma_{\mathcal{M}}) : s'$.
By the lemma B.34, we have that $(\lfloor t' \rfloor_{|\Phi+1|} \cdot \sigma_{\mathcal{M}}) = \lfloor t' \cdot \sigma_{\mathcal{M}} \rfloor_{|\Phi|+1}$.
Thus we get $\mathcal{M};\ \Phi \cdot \sigma_{\mathcal{M}} \vdash \Pi(t_1 \cdot \sigma_{\mathcal{M}}).\ \lfloor t' \cdot \sigma_{\mathcal{M}} \rfloor_{|\Phi|+1} : s'$.
We can now apply the same typing rule to get: $\mathcal{M};\ \Phi \cdot \sigma_{\mathcal{M}} \vdash \lambda(t_1 \cdot \sigma_{\mathcal{M}}).(t_2 \cdot \sigma_{\mathcal{M}}) : \Pi(t_1 \cdot \sigma_{\mathcal{M}}).\ \lfloor t' \cdot \sigma_{\mathcal{M}} \rfloor_{|\Phi|+1}$.
We have $\Pi(t_1 \cdot \sigma_{\mathcal{M}}).\ \lfloor t' \cdot \sigma_{\mathcal{M}} \rfloor_{|\Phi|+1} = \Pi(t_1 \cdot \sigma_{\mathcal{M}}).((\lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma_{\mathcal{M}}) = (\Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|+1}) \cdot \sigma_{\mathcal{M}}$, thus this is the desired result.

Case $\dfrac{\mathcal{M};\ \Phi \vdash t_1 : \Pi(t).t' \qquad \mathcal{M};\ \Phi \vdash t_2 : t}{\mathcal{M};\ \Phi \vdash t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}\ \triangleright$

By induction hypothesis for t_1 we get $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t_1 \cdot \sigma_\mathcal{M} : \Pi(t \cdot \sigma_\mathcal{M}).(t' \cdot \sigma_\mathcal{M})$.
By induction hypothesis for t_2 we get $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t_2 \cdot \sigma_\mathcal{M} : t \cdot \sigma_\mathcal{M}$.
By application of the same typing rule we get $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash (t_1\ t_2) \cdot \sigma_\mathcal{M} : \lceil t' \cdot \sigma_\mathcal{M} \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2 \cdot \sigma_\mathcal{M})$.
We need to prove that $(\lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)) \cdot \sigma_\mathcal{M} = \lceil t' \cdot \sigma_\mathcal{M} \rceil_{|\Phi|} \cdot (\mathsf{id}_{\Phi \cdot \sigma_\mathcal{M}}, t_2 \cdot \sigma_\mathcal{M})$.
From lemma B.35 we have that $(\lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)) \cdot \sigma_\mathcal{M} = (\lceil t' \rceil_{|\Phi|} \cdot \sigma_\mathcal{M}) \cdot ((\mathsf{id}_\Phi, t_2) \cdot \sigma_\mathcal{M})$.
From lemma B.32 we get that this is further equal to: $(\lceil t' \cdot \sigma_\mathcal{M} \rceil_{|\Phi|}) \cdot ((\mathsf{id}_\Phi, t_2) \cdot \sigma_\mathcal{M})$.
From definition of $\cdot$ we get that this is equal to $(\lceil t' \cdot \sigma_\mathcal{M} \rceil_{|\Phi|}) \cdot (\mathsf{id}_\Phi \cdot \sigma_\mathcal{M},\ t_2 \cdot \sigma_\mathcal{M})$.
Last from B.36 we get the desired result.

Case $\dfrac{\mathcal{M}.i = T \qquad T = [\Phi']t' \qquad \mathcal{M};\ \Phi \vdash \sigma : \Phi'}{\mathcal{M};\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\ \triangleright$

Assuming that $\sigma_\mathcal{M}.i = [\Phi'']t$, we need to show that $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t \cdot (\sigma \cdot \sigma_\mathcal{M}) : (t' \cdot \sigma) \cdot \sigma_\mathcal{M}$.
From lemma B.35, we have that $(t' \cdot \sigma) \cdot \sigma_\mathcal{M} = (t' \cdot \sigma_\mathcal{M}) \cdot (\sigma \cdot \sigma_\mathcal{M})$.
So equivalently we need to show $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t \cdot (\sigma \cdot \sigma_\mathcal{M}) : (t' \cdot \sigma_\mathcal{M}) \cdot (\sigma \cdot \sigma_\mathcal{M})$.
Using the second part of the lemma for σ we get: $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash \sigma \cdot \sigma_\mathcal{M} : \Phi' \cdot \sigma_\mathcal{M}$.
From lemma B.38 we get that $\mathcal{M}' \vdash \sigma_\mathcal{M}.i : \mathcal{M}.i \cdot \sigma_\mathcal{M}$.
From hypothesis we have that $\mathcal{M}.i = [\Phi']t'$.
Thus the above typing judgement is rewritten as $\mathcal{M}' \vdash \sigma_\mathcal{M}.i : [\Phi' \cdot \sigma_\mathcal{M}]t \cdot \sigma_\mathcal{M}$.
By inversion we get that $\sigma_\mathcal{M}.i = [\Phi' \cdot \sigma_\mathcal{M}]t$ and that $\mathcal{M}';\ \Phi' \cdot \sigma_\mathcal{M} \vdash t : t' \cdot \sigma_\mathcal{M}$.
** Now we use the main substitution theorem B.22 for t and $\sigma \cdot \sigma_\mathcal{M}$ and get:
$\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t \cdot (\sigma \cdot \sigma_\mathcal{M}) : (t' \cdot \sigma_\mathcal{M}) \cdot (\sigma \cdot \sigma_\mathcal{M})$.

Case (otherwise) $\triangleright$
Simple to prove based on the methods we have shown above.

Part 2 By induction on the typing derivation of σ.

Case $\dfrac{\mathcal{M} \vdash \Phi\ \mathrm{wf}}{\mathcal{M};\ \Phi \vdash \bullet : \bullet}\ \triangleright$ Use of the same typing rule, for $\Phi \cdot \sigma_\mathcal{M}$ which is well formed based on part 3.

Case $\dfrac{\mathcal{M};\ \Phi \vdash \sigma : \Phi' \qquad \mathcal{M};\ \Phi \vdash t : t' \cdot \sigma}{\mathcal{M};\ \Phi \vdash \sigma, t : (\Phi', t')}\ \triangleright$ By induction hypothesis and use of part 1 we get:
$\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash \sigma \cdot \sigma_\mathcal{M} : \Phi' \cdot \sigma_\mathcal{M}$
$\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t \cdot \sigma_\mathcal{M} : (t' \cdot \sigma) \cdot \sigma_\mathcal{M}$
By use of lemma B.35 in the typing for $t \cdot \sigma_\mathcal{M}$ we get that:
$\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash t \cdot \sigma_\mathcal{M} : (t' \cdot \sigma_\mathcal{M}) \cdot (\sigma \cdot \sigma_\mathcal{M})$
By use of the same typing rule we get: $\mathcal{M}';\ \Phi \cdot \sigma_\mathcal{M} \vdash (\sigma \cdot \sigma_\mathcal{M},\ t \cdot \sigma_\mathcal{M}) : (\Phi' \cdot \sigma_\mathcal{M}, t' \cdot \sigma_\mathcal{M})$

Part 3 By induction on the well-formedness derivation of Φ.

Case $\mathcal{M} \vdash \bullet\ \mathrm{wf}\ \triangleright$
Trivial use of the same typing rule.

Case $\dfrac{\mathcal{M} \vdash \Phi\ \mathrm{wf} \qquad \mathcal{M};\ \Phi \vdash t : s}{\mathcal{M} \vdash \Phi,\ t\ \mathrm{wf}}\ \triangleright$
Use of induction hypothesis, part 2, and the same typing rule.

Part 4 By induction on the typing derivation for T.

Case $\dfrac{\mathcal{M};\ \Phi \vdash t : t'}{\mathcal{M} \vdash [\Phi]t : [\Phi]t'}\ \triangleright$
Using part 1 we get $\mathcal{M}'$; $\Phi \cdot \sigma_{\mathcal{M}} \vdash t \cdot \sigma_{\mathcal{M}} : t' \cdot \sigma_{\mathcal{M}}$. Thus using the same typing rule we get $\mathcal{M}' \vdash [\Phi \cdot \sigma_{\mathcal{M}}]t \cdot \sigma_{\mathcal{M}} : [\Phi \cdot \sigma_{\mathcal{M}}]t' \cdot \sigma_{\mathcal{M}}$, which is the desired result.

B.2 Extension with metavariables and polymorphic contexts

In order to incorporate polymorphic contexts, we change the representation of free variables from a deBruijn level to an index into a parametric context. We thus need to redefine the notions of length of a context, variable limits etc. in order to be compatible with the new definition of free variables.

Definition B.40 (Syntax of the language) *The syntax of the logic language is extended below. We use the syntactic class T for modal terms and modal contexts, and the syntactic class K for their classifiers (modal terms and context prefixes). Furthermore, we use a single context* Ψ *for both extensions.*

$$
\begin{array}{rl}
\Phi ::= & \cdots \mid \Phi, X_i \\
\sigma ::= & \cdots \mid \sigma,\ \mathbf{id}(X_i) \\
\Psi ::= & \bullet \mid \Psi,\ K \\
t ::= & s \mid c \mid f_{\mathbf{I}} \mid b_i \mid \lambda(t_1).t_2 \mid t_1\ t_2 \mid \Pi(t_1).t_2 \mid t_1 = t_2 \mid \mathsf{conv}\ t\ t \mid \mathsf{refl}\ t \mid \mathsf{symm}\ t \mid \mathsf{trans}\ t_1\ t_2 \mid \mathsf{congapp}\ t_1\ t_2 \\
& \mid \mathsf{congimpl}\ t_1\ t_2 \mid \mathsf{conglam}\ t \mid \mathsf{congpi}\ t \mid \mathsf{beta}\ t_1\ t_2 \mid X_i/\sigma \\
T ::= & [\Phi]t \mid [\Phi]\Phi' \\
K ::= & [\Phi]t \mid [\Phi]\,\mathsf{ctx} \\
\mathbf{I} ::= & \bullet \mid \mathbf{I},\ \cdot \mid \mathbf{I},\ |X_i|
\end{array}
$$

Definition B.41 (Substitution length) *Redefinition of B.3.*

$\boxed{|\sigma| = \mathbf{I}}$

$$
\begin{array}{lcl}
|\bullet| & = & \bullet \\
|\sigma,\ t| & = & |\sigma|,\ \cdot \\
|\sigma,\ \mathbf{id}(X_i)| & = & |\sigma|,\ |X_i|
\end{array}
$$

Definition B.42 (Ordering of indexes) *We define what it means for an index to be less than another index.*

$\boxed{\mathbf{I} < \mathbf{I}'}$

$$
\begin{array}{lcl}
\mathbf{I} & < & \mathbf{I}',\ \cdot \text{ when } \mathbf{I} = \mathbf{I}' \text{ or } \mathbf{I} < \mathbf{I}' \\
\mathbf{I} & < & \mathbf{I}',\ |X_i| \text{ when } \mathbf{I} = \mathbf{I}' \text{ or } \mathbf{I} < \mathbf{I}'
\end{array}
$$

$\boxed{\mathbf{I} \leq \mathbf{I}'}$

$$
\mathbf{I} \ \leq\ \mathbf{I}' \text{ when } \mathbf{I} = \mathbf{I}' \text{ or } \mathbf{I} < \mathbf{I}'
$$

Definition B.43 (Substitution access) *Redefinition of B.4. We assume* $\mathbf{I} < |\sigma|$.

$\boxed{\sigma.\mathbf{I}}$

$$\begin{array}{lcl}(\sigma,\, t).\mathbf{I} & = & t \text{ when } |\sigma| = \mathbf{I} \\ (\sigma,\, t).\mathbf{I} & = & \sigma.\mathbf{I} \text{ otherwise} \\ (\sigma,\, \mathbf{id}(X_i)).\mathbf{I} & = & t \text{ when } |\sigma| = \mathbf{I} \\ (\sigma,\, \mathbf{id}(X_i)).\mathbf{I} & = & \sigma.\mathbf{I} \text{ otherwise}\end{array}$$

Definition B.44 (Context length and access) *Redefinition of context length and context access, from definition B.2. Furthermore we define length and element access for environments of contexts. Element access assumes* $\mathbf{I} < |\Phi|$.

$\boxed{|\Phi| = \mathbf{I}}$

$$\begin{array}{lcl}|\bullet| & = & \bullet \\ |\Phi,\, t| & = & |\Phi|,\, \cdot \\ |\Phi,\, X_i| & = & |\Phi|,\, |X_i|\end{array}$$

$\boxed{\Phi.\mathbf{I}}$

$$\begin{array}{lcl}(\Phi,\, t).\mathbf{I} & = & t \text{ when } |\Phi| = \mathbf{I} \\ (\Phi,\, t).\mathbf{I} & = & \Phi.\mathbf{I} \text{ otherwise} \\ (\Phi,\, X_i).\mathbf{I} & = & X_i \text{ when } |\Phi| = \mathbf{I} \\ (\Phi,\, X_i).\mathbf{I} & = & \Phi.\mathbf{I} \text{ otherwise}\end{array}$$

Definition B.45 (Extensions context length and access) *New definition.*

$\boxed{|\Psi|}$

$$\begin{array}{lcl}|\bullet| & = & 0 \\ |\Psi,\, K| & = & |\Psi| + 1\end{array}$$

$\boxed{\Psi.i}$

$$\begin{array}{lcl}(\Psi,\, K).|\Psi| & = & K \\ (\Psi,\, K).i & = & \Psi.i \text{ when } i < |\Psi|\end{array}$$

Definition B.46 (Substitution application) *Extension of substitution application from definition B.5. The application of a substitution to a term is entirely identical as before, with a slight adjustment for the new definitions of variable indexes.*

$\boxed{t \cdot \sigma}$

$$f_{\mathbf{I}} \cdot \sigma \;=\; \sigma.\mathbf{I}$$

$\boxed{\sigma' \cdot \sigma}$

$$(\sigma',\, \mathbf{id}(X_i)) \cdot \sigma \;=\; \sigma' \cdot \sigma,\, \mathbf{id}(X_i)$$

Definition B.47 (Identity substitution) *Redefinition of identity substitution from B.6.*

$$\begin{array}{lcl} \mathsf{id}_{\bullet} & = & \bullet \\ \mathsf{id}_{\Phi,\,t} & = & \mathsf{id}_{\Phi},\, f_{|\Phi|} \\ \mathsf{id}_{\Phi,\,X_i} & = & \mathsf{id}_{\Phi},\, \mathbf{id}(X_i) \end{array}$$

Definition B.48 (Variable limits for terms and substitutions) *Redefinition of the definition B.7.*

$\boxed{t <^f \mathbf{I}}$

$$\begin{array}{lcll} s <^f \mathbf{I} & & & \\ c <^f \mathbf{I} & & & \\ f_{\mathbf{I}} <^f \mathbf{I}' & \Leftarrow & \mathbf{I} < \mathbf{I}' & \\ b_i <^f \mathbf{I} & & & \\ (\lambda(t_1).t_2) <^f \mathbf{I} & \Leftarrow & t_1 <^f \mathbf{I} \wedge t_2 <^f \mathbf{I} & \\ t_1\, t_2 <^f \mathbf{I} & \Leftarrow & t_1 <^f \mathbf{I} \wedge t_2 <^f \mathbf{I} & \\ & & \cdots & \end{array}$$

$\boxed{\sigma <^f \mathbf{I}}$

$$\begin{array}{lcl} \bullet <^f \mathbf{I} & & \\ \sigma,\, t <^f \mathbf{I} & \Leftarrow & \sigma <^f \mathbf{I} \wedge t <^f \mathbf{I} \\ \sigma,\, \mathbf{id}(X_i) <^f \mathbf{I} & \Leftarrow & \sigma <^f \mathbf{I} \wedge \exists \mathbf{I}' : (\mathbf{I}',\, |X_i|) \leq \mathbf{I} \end{array}$$

$\boxed{\sigma <^b n}$

$$\sigma,\, \mathbf{id}(\phi_i) <^b n \quad \Leftarrow \quad \sigma <^b n$$

Definition B.49 (Extension of freshening) *This is an extension of definition B.8 and adjustment for indexes. We assume $t <^f \mathbf{I}$ and $\sigma <^f \mathbf{I}$. Also $t <^b n+1$ and $\sigma <^b n+1$.*

$\boxed{\lceil t \rceil^n_{\mathbf{I}}}$

$$\begin{array}{lcl} \lceil b_n \rceil^n_{\mathbf{I}} & = & f_{\mathbf{I}} \\ \lceil b_i \rceil^n_{\mathbf{I}} & = & b_i \end{array}$$

$\boxed{\lceil \sigma \rceil^n_{\mathbf{I}}}$

$$\begin{array}{lcl} \lceil \bullet \rceil^n_{\mathbf{I}} & = & \bullet \\ \lceil \sigma,\, t \rceil^n_{\mathbf{I}} & = & \lceil \sigma \rceil^n_{\mathbf{I}},\, \lceil t \rceil^n_{\mathbf{I}} \\ \lceil \sigma,\, \mathbf{id}(X_i) \rceil^n_{\mathbf{I}} & = & \lceil \sigma \rceil^n_{\mathbf{I}},\, \mathbf{id}(X_i) \end{array}$$

Definition B.50 (Extension of binding) *This is an extension of definition B.9 and adjustment for indexes. We assume $t <^f \mathbf{I}$ and $\sigma <^f \mathbf{I}$. Also $t <^b n$ and $\sigma <^b n$.*

$\boxed{\lfloor t \rfloor^n_{\mathbf{I}}}$

$$\begin{array}{lcl} \lfloor f_{\mathbf{I}'} \rfloor^n_{\mathbf{I}} & = & b_n \text{ when } \mathbf{I} = \mathbf{I}', \cdot \\ \lfloor f_{\mathbf{I}'} \rfloor^n_{\mathbf{I}} & = & f_{\mathbf{I}'} \text{ otherwise} \end{array}$$

$\boxed{\lfloor \sigma \rfloor^n_{\mathbf{I}}}$

$$\begin{array}{lcl} \lfloor \bullet \rfloor^n_{\mathbf{I}} & = & \bullet \\ \lfloor \sigma, t \rfloor^n_{\mathbf{I}} & = & \lfloor \sigma \rfloor^n_{\mathbf{I}}, \lfloor t \rfloor^n_{\mathbf{I}} \\ \lfloor \sigma, \mathbf{id}(X_i) \rfloor^n_{\mathbf{I}} & = & \lfloor \sigma \rfloor^n_{\mathbf{I}}, \mathbf{id}(X_i) \end{array}$$

Definition B.51 (Environment subsumption) *We define what it means for an environment to be a subenvironment (be a prefix of; or be subsumed by) another one.*

$\boxed{\Phi \subseteq \Phi'}$

$$\begin{array}{lcl} \Phi \subseteq \Phi & & \\ \Phi \subseteq \Phi', t & \Leftarrow & \Phi \subseteq \Phi' \\ \Phi \subseteq \Phi', X_i & \Leftarrow & \Phi \subseteq \Phi' \end{array}$$

$\boxed{\Psi \subseteq \Psi'}$

$$\begin{array}{lcl} \Psi \subseteq \Psi & & \\ \Psi \subseteq \Psi', K & \Leftarrow & \Psi \subseteq \Psi' \end{array}$$

Definition B.52 (Substitution subsumption) *We define what it means for an substitution to be a prefix of another one.*

$\boxed{\sigma \subseteq \sigma'}$

$$\begin{array}{lcl} \sigma \subseteq \sigma & & \\ \sigma \subseteq \sigma', t & \Leftarrow & \sigma \subseteq \sigma' \\ \sigma \subseteq \sigma', \mathbf{id}(X_i) & \Leftarrow & \sigma \subseteq \sigma' \end{array}$$

Definition B.53 *The typing judgements defined in B.10 and are redefined as follows.*

1. $\vdash \Sigma$ *wf is adjusted as shown below.*
2. $\vdash_\Sigma \Phi$ *wf is redefined as* $\Psi \vdash_\Sigma \Phi$ *wf, and the rules below are added.*
3. $\Phi \vdash t : t'$ *is redefined as* $\Psi;\ \Phi \vdash t : t'$*, and adjusted as shown below.*
4. $\Phi \vdash \sigma : \Phi'$ *is redefined as* $\Psi;\ \Phi \vdash \sigma : \Phi'$ *and the rules below are added.*
5. $\vdash \Psi$ *wf is defined below.*
6. $\Psi \vdash T : K$ *is defined below.*

$\boxed{\vdash \Sigma \text{ wf}}$

$$\frac{\vdash \Sigma \text{ wf} \qquad \bullet;\, \bullet \vdash t : s \qquad (c :) \notin \Sigma}{\vdash (\Sigma,\, c : t) \text{ wf}}$$

$\boxed{\Psi \vdash_\Sigma \Phi \text{ wf}}$

$$\frac{}{\Psi \vdash \bullet \text{ wf}} \qquad \frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi;\, \Phi \vdash t : s}{\Psi \vdash (\Phi,\, t) \text{ wf}} \qquad \frac{\Psi \vdash \Phi \text{ wf} \qquad \Psi.i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash (\Phi,\, X_i) \text{ wf}}$$

$\boxed{\Psi;\, \Phi \vdash t : t'}$

$$\frac{c : t \in \Sigma}{\Psi;\, \Phi \vdash_\Sigma c : t} \qquad \frac{\Phi.\mathbf{I} = t}{\Psi;\, \Phi \vdash f_{\mathbf{I}} : t} \qquad \frac{\Psi;\, \Phi \vdash t_1 : s \qquad \Psi;\, \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi;\, \Phi \vdash \Pi(t_1).t_2 : s''}$$

$$\frac{\Psi;\, \Phi \vdash t_1 : s \qquad \Psi;\, \Phi,\, t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi;\, \Phi \vdash \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|,\,\cdot} : s'}{\Psi;\, \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|,\,\cdot}} \qquad \frac{\Psi;\, \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\, \Phi \vdash t_2 : t}{\Psi;\, \Phi \vdash t_1\, t_2 : \lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}$$

$$\frac{\Psi.i = T \qquad T = [\Phi']\,t' \qquad \Psi;\, \Phi \vdash \sigma : \Phi'}{\Psi;\, \Phi \vdash X_i/\sigma : t' \cdot \sigma}$$

$\boxed{\Psi;\, \Phi \vdash \sigma : \Phi'}$

$$\frac{}{\Psi;\, \Phi \vdash \bullet : \bullet} \qquad \frac{\Psi;\, \Phi \vdash \sigma : \Phi' \qquad \Psi;\, \Phi \vdash t : t' \cdot \sigma}{\Psi;\, \Phi \vdash (\sigma,\, t) : (\Phi',\, t')} \qquad \frac{\Psi;\, \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\, X_i \subseteq \Phi}{\Psi;\, \Phi \vdash (\sigma,\, \mathbf{id}(X_i)) : (\Phi',\, X_i)}$$

$\boxed{\vdash \Psi \text{ wf}}$

$$\frac{}{\vdash \Psi \text{ wf}} \qquad \frac{\vdash \Psi \text{ wf} \qquad \Psi \vdash \Phi \text{ wf}}{\vdash (\Psi,\, [\Phi]\,\mathsf{ctx}) \text{ wf}} \qquad \frac{\vdash \Psi \text{ wf} \qquad \Psi \vdash [\Phi]\,t : [\Phi]\,s}{\vdash (\Psi,\, [\Phi]\,t) \text{ wf}}$$

$\boxed{\Psi \vdash T : K}$

$$\frac{\Psi;\, \Phi \vdash t : t'}{\Psi \vdash [\Phi]\,t : [\Phi]\,t'} \qquad \frac{\Psi \vdash \Phi,\, \Phi' \text{ wf}}{\Psi \vdash [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}}$$

Lemma B.54 (Extension of lemma 2) *1. If $t <^f \mathbf{I}$ and $|\Phi| = \mathbf{I}$ then $t \cdot id_\Phi = t$.*

2. If $\sigma <^f \mathbf{I}$ and $|\Phi| = \mathbf{I}$ then $\sigma \cdot id_\Phi = \sigma$.

Part 1 is proved by induction on $t <^f \mathbf{I}$. The interesting case is $f_{\mathbf{I}'}$, with $\mathbf{I}' < \mathbf{I}$. In this case we have to prove $\mathsf{id}_\Phi.\mathbf{I}' = f_{\mathbf{I}'}$. This is done by induction on $\mathbf{I}' < \mathbf{I}$.
When $\mathbf{I} = \mathbf{I}'$, $\cdot$ we have by inversion of $|\Phi| = \mathbf{I}$ that $\Phi = \Phi'$, t and $|\Phi'| = \mathbf{I}'$. Thus $\mathsf{id}_\Phi = \mathsf{id}\Phi'$, $f_{\mathbf{I}'}$ and thus the desired result.
When $\mathbf{I} = \mathbf{I}'$, $|X_i|$, exactly as above.
When $\mathbf{I} = \mathbf{I}^*$, $\cdot$ and $\mathbf{I}' < \mathbf{I}^*$, we have that $\Phi = \Phi^*$, t and $|\Phi^*| = \mathbf{I}^*$. By (inner) induction hypothesis we get that $\mathsf{id}_{\Phi^*}.\mathbf{I}' = f_{\mathbf{I}'}$. From this directly we get that $\mathsf{id}_\Phi.\mathbf{I}' = f_{\mathbf{I}'}$.
When $\mathbf{I} = \mathbf{I}^*$, $|X_i|$ and $\mathbf{I}' < \mathbf{I}^*$, entirely as the previous case.
Part 2 is trivial to prove by induction and use of part 1 in cases $\sigma = \bullet$ or $\sigma = \sigma'$, t. In the case $\sigma = \sigma'$, $\mathbf{id}(X_i)$ we have: $\sigma' <^f \mathbf{I}$ thus by induction $\sigma' \cdot \mathsf{id}_\Phi = \sigma'$, and furthermore $(\sigma',\, \mathbf{id}(X_i)) \cdot \mathsf{id}_\Phi = \sigma$.

Lemma B.55 (Length of subcontexts) *If* $\Phi \subseteq \Phi'$ *then* $|\Phi| \leq |\Phi'|$.

Trivial by induction on $\Phi \subseteq \Phi'$.

Lemma B.56 (Variable limits can be increased) *1. If* $t <^f \mathbf{I}$ *and* $\mathbf{I} < \mathbf{I}'$ *then* $t <^f \mathbf{I}'$

2. If $t <^b n$ *and* $n < n'$ *then* $t <^b n'$

3. If $\sigma <^f \mathbf{I}$ *and* $\mathbf{I} < \mathbf{I}'$ *then* $\sigma <^f \mathbf{I}'$

4. If $\sigma <^b n$ *and* $n < n'$ *then* $\sigma <^b n'$

Trivial by induction on t or σ.

Lemma B.57 (Extension of lemma 2) *1. If* $\Psi;\ \Phi \vdash t : t'$ *then* $t <^f |\Phi|$ *and* $t <^b 0$.

2. If $\Psi;\ \Phi \vdash \sigma : \Phi'$ *then* $\sigma <^f |\Phi|$, $\sigma <^b 0$ *and* $|\sigma| = |\Phi'|$.

Part 1 is proved similarly as before.
Part 2 needs to account for the new case $\sigma = \sigma^*,\ \mathbf{id}(X_i)$.
By inversion of typing for σ we get that $\Phi' = \Phi^*,\ X_i$ with $\sigma^* : \Phi^*$. By induction we get that $\sigma^* <^f |\Phi^*|$. Again by inversion of typing for σ we get that $\Phi^*,\ X_i \subseteq \Phi$. Thus $\sigma^* <^f |\Phi|$ by use of lemma B.56. Furthermore from $\Phi^*,\ X_i \subseteq \Phi$ and lemma B.55 we get that $|\Phi^*|,\ |X_i| \leq |\Phi|$. Thus for $\mathbf{I}' = |\Phi^*|$ we have $\mathbf{I}',\ |X_i| < |\Phi|$ thus we overall get $\sigma <^f |\Phi|$.
Furthermore the other two parts of the theorem are trivial from induction hypothesis.

Lemma B.58 (Extension of lemma B.13) *If* $\Psi \vdash \Phi$ *wf then for any* Φ' *such that* $\Phi \subseteq \Phi'$ *and* $\Psi \vdash \Phi'$ *wf, we have that* $\Psi;\ \Phi' \vdash id_\Phi : \Phi$.

Similar to the original proof. The new case for $\Phi = \Phi',\ X_i$ works as follows. By induction hypothesis for Φ' we get that $\Psi;\ \Phi',\ X_i \vdash \mathsf{id}_{\Phi'} : \Phi'$. Now for any environment Φ^* such that $\Phi',\ X_i \subseteq \Phi^*$, by using the typing rule for $\mathbf{id}(X_i)$, we get the desired.

Lemma B.59 (Extension of lemma B.14) *If* $\Psi \vdash \Phi$ *wf and* $|\Phi| = \mathbf{I}$ *then for all* $\mathbf{I}' < \mathbf{I}$ *with* $\Phi.\mathbf{I}' = t$, *we have* $\Phi.\mathbf{I}' <^f \mathbf{I}$.

Identical as before.

Lemma B.60 (Extension of lemmas B.15 and B.15) *1. If* $t <^f \mathbf{I},\ |\sigma| = \mathbf{I},\ t \cdot \sigma = t'$ *and* $\sigma \subseteq \sigma'$ *then* $t \cdot \sigma' = t'$.

2. If $\sigma <^f \mathbf{I},\ |\sigma'| = \mathbf{I},\ \sigma \cdot \sigma' = \sigma_r$ *and* $\sigma \subseteq \sigma''$ *then* $\sigma \cdot \sigma'' = \sigma_r$.

Part 1 is identical as before. In part 2, in case $\sigma = \sigma,\ \mathbf{id}(X_i)$, proved trivially by definition of substitution application.

Lemma B.61 (Extension of lemma B.16) *If* $\Psi \vdash \Phi$ *wf,* $\Phi.\mathbf{I} = t$ *and* $\Psi;\ \Phi' \vdash \sigma : \Phi$, *then* $\Psi;\ \Phi' \vdash \sigma.\mathbf{I} : t \cdot \sigma$.

The proof proceeds by structural induction on the typing derivation for σ as before. In case $\sigma = \sigma^*,\ \mathbf{id}(X_i)$, we have that $(\Phi^*,\ X_i) \subseteq \Phi'$. We have that $\Phi^*.\mathbf{I} = \Phi.\mathbf{I} = t$ (since $\mathbf{I} \not= |\Phi^*|$, because $(\Phi^*,\ X_i).|\Phi| \not= t$). Thus from induction hypothesis for σ^* we get that $\Psi;\ \Phi' \vdash \sigma^*.\mathbf{I} : t \cdot \sigma^*$. Using lemma B.60 and also the fact that $\sigma.\mathbf{I} = \sigma^*.\mathbf{I}$, we get that $\Psi;\ \Phi' \vdash \sigma.\mathbf{I} : t \cdot \sigma$.

Lemma B.62 (Extension of lemma B.17) *1. If* $t <^f \mathbf{I},\ t <^b n+1,\ \sigma <^f \mathbf{I}'$ *and* $|\sigma| = \mathbf{I}$ *then* $\lceil t \cdot \sigma \rceil^n_{\mathbf{I}'} = \lceil t \rceil^n_{\mathbf{I}} \cdot (\sigma, f_{\mathbf{I}'})$.

2. If $\sigma' <^f \mathbf{I}$, $\sigma' <^b n+1$, $\sigma <^f \mathbf{I}'$ *and* $|\sigma| = \mathbf{I}$ *then* $\lceil \sigma' \cdot \sigma \rceil^n_{\mathbf{I}'} = \lceil \sigma' \rceil^n_{\mathbf{I}} \cdot (\sigma, f_{\mathbf{I}'})$.

Part 1 is entirely similar as before, with slight adjustments to account for the new type of indices. Part 2 needs to account for the new case of $\sigma' = \sigma'', \mathbf{id}(X_i)$, which is entirely trivial based on the definition.

Lemma B.63 (Extension of lemma B.18) *1. If* $t <^f \mathbf{I}$, $\cdot$, $t <^b n$, $\sigma <^f \mathbf{I}'$ *and* $|\sigma| = \mathbf{I}$ *then* $\lfloor t \cdot (\sigma, f_{\mathbf{I}'}) \rfloor^n_{\mathbf{I}',\cdot} = \lfloor t \rfloor^n_{\mathbf{I}',\cdot} \cdot \sigma$.

2. If $\sigma' <^f \mathbf{I}$, $\cdot$, $\sigma' <^b n$, $\sigma <^f \mathbf{I}'$ *and* $|\sigma| = m$ *then* $\lfloor \sigma' \cdot (\sigma, f_{\mathbf{I}'}) \rfloor^n_{\mathbf{I}',\cdot} = \lfloor \sigma' \rfloor^n_{\mathbf{I}',\cdot} \cdot \sigma$.

Similarly to the above.

Lemma B.64 (Extension of lemma B.19) *1. If* $t <^f \mathbf{I}$, $|\sigma| = \mathbf{I}$, $\sigma <^f \mathbf{I}'$ *and* $|\sigma'| = \mathbf{I}'$ *then* $(t \cdot \sigma) \cdot \sigma' = t \cdot (\sigma \cdot \sigma')$.

2. If $\sigma_1 <^f \mathbf{I}$, $|\sigma| = \mathbf{I}$, $\sigma <^f \mathbf{I}'$ *and* $|\sigma'| = \mathbf{I}'$ *then* $(\sigma_1 \cdot \sigma) \cdot \sigma' = \sigma_1 \cdot (\sigma \cdot \sigma')$.

Part 1 is identical as before. Part 2 needs to account for the case where $\sigma_1 = \sigma'_1, \mathbf{id}(X_i)$, which is entirely trivial.

Lemma B.65 (Extension of lemma B.20) *If* $|\sigma| = \mathbf{I}$ *and* $|\Phi| = \mathbf{I}$ *then* $id_\Phi \cdot \sigma = \sigma$.

We need to account for the new case of $\Phi = \Phi', X_i$. In that case, $\mathsf{id}_{\Phi', X_i} = \mathsf{id}_{\Phi'}, \mathbf{id}(X_i)$. By inversion of $|\sigma| = \mathbf{I} = |\Phi'|, |X_i|$ we get that $\sigma = \sigma', \mathbf{id}(X_i)$. By induction hypothesis we get $\mathsf{id}_{\Phi'} \cdot \sigma' = \sigma'$. By lemma B.60 we get $\mathsf{id}_{\Phi'} \cdot \sigma = \sigma'$. Last it is trivial to see that $(\mathsf{id}_{\Phi'}, \mathbf{id}(X_i)) \cdot \sigma = \sigma', \mathbf{id}(X_i) = \sigma$.

Lemma B.66 (Extension of lemma B.21) *1. If* $\lceil t \rceil^n_{\mathbf{I}} = \lceil t' \rceil^n_{\mathbf{I}}$ *then* $t = t'$.

2. If $\lceil \sigma \rceil^n_{\mathbf{I}} = \lceil \sigma' \rceil^n_{\mathbf{I}}$ *then* $\sigma = \sigma'$.

Part 1 is identical as before; part 2 holds trivially for the new case of σ.

Theorem B.67 (Extension of main substitution theorem B.22) *1. If* $\Psi;\ \Phi \vdash t : t'$ *and* $\Psi;\ \Phi' \vdash \sigma : \Phi$ *then* $\Psi;\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma$.

2. If $\Psi;\ \Phi' \vdash \sigma : \Phi$ *and* $\Psi;\ \Phi'' \vdash \sigma' : \Phi'$ *then* $\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi$.

3. If $\Psi \vdash [\Phi']t : [\Phi']t'$ *and* $\Psi;\ \Phi \vdash \sigma : \Phi'$ *then* $\Psi \vdash [\Phi]t \cdot \sigma : [\Phi]t' \cdot \sigma$.

Part 1 is identical as before; all the needed theorems were adjusted above, so the new form of indexes does not change the proof at all. The only case that needs adjustment is the metavariables case.

$$\textbf{Case}\ \frac{\Psi.i = T \qquad T = [\Phi']t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma_0 : t'}\ \triangleright$$

From $\Psi;\ \Phi \vdash X_i/\sigma_0 : t'$ we get that $\Psi.i = [\Phi_0]t_0$, $\Psi;\ \Phi \vdash \sigma_0 : \Phi_0$ and $t' = t_0 \cdot \sigma_0$.
Applying the second part of the lemma for $\sigma = \sigma_0$ and $\sigma' = \sigma$ we get that $\Psi;\ \Phi' \vdash \sigma_0 \cdot \sigma' : \Phi_0$.
Thus applying the same typing rule for $t = X_i/(\sigma_0 \cdot \sigma)$ we get that $\Psi;\ \Phi' \vdash X_i/(\sigma_0 \cdot \sigma') : t_0 \cdot (\sigma_0 \cdot \sigma')$.
Taking into account the definition of $\cdot$ and also lemma B.64, we have that this is the desired result.

For the second part, we need to account for the new case of substitutions.

$$\textbf{Case}\ \frac{\Psi;\ \Phi' \vdash \sigma : \Phi_0 \qquad \Psi.i = [\Phi_0]\,\mathsf{ctx} \qquad \Phi_0,\, X_i \subseteq \Phi'}{\Psi;\ \Phi' \vdash (\sigma,\, \mathbf{id}(X_i)) : (\Phi_0,\, X_i)}\ \triangleright$$

By induction hypothesis for σ, we get: $\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi_0$.
We need to prove that $(\Phi_0,\, X_i) \subseteq \Phi''$.

We have that $\Psi;\ \Phi'' \vdash \sigma' : \Phi'$.
By induction on $(\Phi_0,\ X_i) \subseteq \Phi'$ and repeated inversions of σ' we arrive at a $\sigma'' \subseteq \sigma'$ such that:
$\Psi;\ \Phi'' \vdash \sigma'' : \Phi_0,\ X_i$
By inversion of this we get that $(\Phi_0,\ X_i) \subseteq \Phi''$.
Thus, using the same typing rule, we get $\Psi;\ \Phi'' \vdash (\sigma \cdot \sigma', \mathbf{id}(X_i)) : (\Phi_0,\ X_i)$, which is the desired.

For the third part, the proof is identical as before.

Lemma B.68 (Extension of lemma B.24) *If $\Psi;\ \Phi \vdash t : t'$ then either $t' = \mathsf{Type}'$ or $\Psi;\ \Phi \vdash t' : s$.*

Identical as before.

Lemma B.69 (Extension of the lemma B.25) *1. If $\Psi;\ \Phi \vdash t : t'$ and $\Phi \subseteq \Phi'$ then $\Psi;\ \Phi' \vdash t : t'$.*
2. If $\Psi;\ \Phi \vdash \sigma : \Phi''$ and $\Phi \subseteq \Phi'$ then $\Psi;\ \Phi' \vdash \sigma : \Phi''$.

Identical as before.

Lemma B.70 (Adaptation of lemma 4) *1. If $\Psi;\Phi \vdash t : t'$ and $\Psi \subseteq \Psi'$ then $\Psi';\ \Phi \vdash t : t'$.*
2. If $\Psi;\Phi \vdash \sigma : \Phi'$ and $\Psi \subseteq \Psi'$ then $\Psi';\ \Phi \vdash \sigma : \Phi'$.
3. If $\Psi \vdash \Phi$ wf and $\Psi \subseteq \Psi'$ then $\Psi' \vdash \Phi$ wf.
4. If $\Psi \vdash T : K$ and $\Psi \subseteq \Psi'$ then $\Psi' \vdash T : K$.

Parts 2 and 3 are trivial for the new cases; otherwise identical as before.

Now we have proved the fundamentals. We proceed to define substitutions for the extension variables (meta- and context-variables), typing for such substitutions, and prove an extensions substitution theorem.

Definition B.71 (Substitutions of extension variables) *The syntax of substitutions for meta- and context-variables is given below.*

$$\sigma_\Psi ::= \bullet \mid \sigma_\Psi,\ T$$

Definition B.72 (Context, substitution, index concatenation) *We define what it means to concatenate one context (substitution, index) to another.*

$\boxed{\Phi,\ \Phi'}$

$$\begin{array}{lcl} \Phi,\ (\bullet) & = & \Phi \\ \Phi,\ (\Phi',\ t) & = & (\Phi,\ \Phi'),\ t \\ \Phi,\ (\Phi',\ X_i) & = & (\Phi,\ \Phi'),\ X_i \end{array}$$

$\boxed{\sigma,\ \sigma'}$

$$\begin{array}{lcl} \sigma,\ (\bullet) & = & \sigma \\ \sigma,\ (\sigma',\ t) & = & (\sigma,\ \sigma'),\ t \\ \sigma,\ (\sigma',\ \mathbf{id}(X_i)) & = & (\sigma,\ \sigma'),\ \mathbf{id}(X_i) \end{array}$$

$\boxed{I,\, I'}$

$$\begin{array}{lcl} \mathbf{I},\, (\bullet) & = & \mathbf{I} \\ \mathbf{I},\, (\mathbf{I}',\, \cdot) & = & (\mathbf{I},\, \mathbf{I}'),\, \cdot \\ \mathbf{I},\, (\mathbf{I}',\, |X_i|) & = & (\mathbf{I},\, \mathbf{I}'),\, |X_i| \end{array}$$

Definition B.73 (Partial identity substitution) *We define what partial identity substitutions (for a suffix of a context) are.*

$\boxed{\mathsf{id}_{[\Phi]\,\Phi'}}$

$$\begin{array}{lcl} \mathsf{id}_{[\Phi]\,\bullet} & = & \bullet \\ \mathsf{id}_{[\Phi]\,\Phi',\,t} & = & \mathsf{id}_{[\Phi]\,\Phi'},\, f_{|\Phi|+|\Phi'|} \\ \mathsf{id}_{[\Phi]\,\Phi',\,X_i} & = & \mathsf{id}_{[\Phi]\,\Phi'},\, \mathbf{id}(X_i) \end{array}$$

Definition B.74 (Extensions substitution length and access) *Defined below.*

$\boxed{|\sigma_\Psi|}$

$$\begin{array}{lcl} |\bullet| & = & 0 \\ |\sigma_\Psi,\, T| & = & 1+|\sigma_\Psi| \end{array}$$

Definition B.75 (Extension substitution and context concatenation) *We define concatenation of extension substitutions and extensions contexts below.*

$\boxed{\Psi,\, \Psi'}$

$$\begin{array}{lcl} \Psi,\, (\bullet) & = & \Psi \\ \Psi,\, (\Psi',\, K) & = & (\Psi,\, \Psi'),\, K \end{array}$$

$\boxed{\sigma_\Psi,\, \sigma'_\Psi}$

$$\begin{array}{lcl} \sigma_\Psi,\, (\bullet) & = & \sigma_\Psi \\ \sigma_\Psi,\, (\sigma'_\Psi,\, T) & = & (\sigma_\Psi,\, \sigma'_\Psi),\, T \end{array}$$

Definition B.76 (Extensions substitution subsumption) *Defined below.*

$\boxed{\sigma_\Psi \subseteq \sigma'_\Psi}$

$$\begin{array}{l} \sigma_\Psi \subseteq \sigma_\Psi \\ \sigma_\Psi \subseteq \sigma'_\Psi,\, T \quad \Leftarrow \quad \sigma_\Psi \subseteq \sigma'_\Psi \end{array}$$

Definition B.77 (Application of extensions substitution) *This is an adaptation of definition B.28.*

$\boxed{\mathbf{I}\cdot\sigma\Psi}$

$$
\begin{array}{rlcl}
 & \bullet\cdot\sigma\Psi & = & \bullet \\
 & (\mathbf{I},\,\cdot)\cdot\sigma\Psi & = & (\mathbf{I}\cdot\sigma\Psi),\,\cdot \\
* & (\mathbf{I},\,|X_i|)\cdot\sigma\Psi & = & (\mathbf{I}\cdot\sigma\Psi),\,|\Phi'| \text{ when } \sigma\Psi.i = [\Phi]\Phi'
\end{array}
$$

$\boxed{t\cdot\sigma\Psi}$

$$
\begin{array}{rcl}
f_{\mathbf{I}}\cdot\sigma\Psi & = & f_{\mathbf{I}\cdot\sigma\Psi} \\
(X_i/\sigma)\cdot\sigma\Psi & = & t\cdot(\sigma\cdot\sigma\Psi) \text{ when } \sigma\Psi.i = [\Phi]t
\end{array}
$$

$\boxed{\sigma\cdot\sigma\Psi}$

$$
\begin{array}{rlcl}
 & \bullet\cdot\sigma\Psi & = & \bullet \\
 & (\sigma,\,t)\cdot\sigma\Psi & = & \sigma\cdot\sigma\Psi,\,t\cdot\sigma\Psi \\
* & (\sigma,\,\mathbf{id}(X_i))\cdot\sigma\Psi & = & \sigma\cdot\sigma\Psi,\,\mathsf{id}_{\sigma\Psi.i} \text{ when } \sigma\Psi.i = [\Phi]\Phi'
\end{array}
$$

$\boxed{\Phi\cdot\sigma\Psi}$

$$
\begin{array}{rcl}
\bullet\cdot\sigma\Psi & = & \bullet \\
(\Phi,\,t)\cdot\sigma\Psi & = & \Phi\cdot\sigma\Psi,\,t\cdot\sigma\Psi \\
(\Phi,\,X_i)\cdot\sigma\Psi & = & \Phi\cdot\sigma\Psi,\,\Phi' \text{ when } \sigma\Psi.i = [\Phi]\Phi'
\end{array}
$$

$\boxed{T\cdot\sigma\Psi}$

$$
\begin{array}{rcl}
([\Phi]t)\cdot\sigma\Psi & = & [\Phi\cdot\sigma\Psi]\,(t\cdot\sigma\Psi) \\
([\Phi]\Phi')\cdot\sigma\Psi & = & [\Phi\cdot\sigma\Psi](\Phi'\cdot\sigma\Psi)
\end{array}
$$

$\boxed{K\cdot\sigma\Psi}$

$$
\begin{array}{rcl}
([\Phi]t)\cdot\sigma\Psi & = & [\Phi\cdot\sigma\Psi]\,(t\cdot\sigma\Psi) \\
([\Phi]\,\mathsf{ctx})\cdot\sigma\Psi & = & [\Phi\cdot\sigma\Psi]\,\mathsf{ctx}
\end{array}
$$

$\boxed{\sigma\Psi\cdot\sigma'_\Psi}$

$$
\begin{array}{rcl}
\bullet\cdot\sigma'_\Psi & = & \bullet \\
(\sigma\Psi,\,T)\cdot\sigma'_\Psi & = & \sigma\Psi\cdot\sigma'_\Psi,\,T\cdot\sigma'_\Psi
\end{array}
$$

Definition B.78 (Application of extended substitution to open extended context) *Assuming that Ψ' does not include variables bigger than $X_{|\Psi|}$, we have:*

$\boxed{\Psi'\cdot\sigma\Psi}$

$$
\begin{array}{rcl}
\bullet\cdot\sigma\Psi & = & \bullet \\
(\Psi',\,K)\cdot\sigma\Psi & = & \Psi'\cdot\sigma\Psi,\,K\cdot(\sigma\Psi,\,X_{|\Psi|},\,\cdots,X_{|\Psi|+|\Psi'|})
\end{array}
$$

Definition B.79 (Identity extension substitution) *The identity substitution for extension contexts is defined below.*

$\boxed{\mathsf{id}_\Psi}$

$$\begin{array}{lcl} \mathsf{id}_\bullet & = & \bullet \\ \mathsf{id}_{\Psi,\,K} & = & \mathsf{id}_\Psi,\, X_{|\Psi|} \end{array}$$

Definition B.80 (Extensions substitution typing) *The typing judgement for extensions substitutions is redefined as* $\Psi \vdash \sigma_\Psi : \Psi'$. *The rules are given below. We also define typing for open extension contexts.*

$\boxed{\Psi \vdash \sigma_\Psi : \Psi'}$

$$\frac{}{\Psi \vdash \bullet : \bullet} \qquad \frac{\Psi \vdash \sigma_\Psi : \Psi' \qquad \Psi \vdash T : K \cdot \sigma_\Psi}{\Psi \vdash (\sigma_\Psi,\, T) : (\Psi',\, K)}$$

$\boxed{\Psi \vdash \Psi' \text{ wf}}$

$$\frac{\vdash \Psi,\, \Psi' \text{ wf}}{\Psi \vdash \Psi' \text{ wf}}$$

Lemma B.81 (Interaction of extensions substitution and length) *1.* $|\sigma| \cdot \sigma_\Psi = |\sigma \cdot \sigma_\Psi|$
2. $|\Phi| \cdot \sigma_\Psi = |\Phi \cdot \sigma_\Psi|$

By induction on σ and Φ.

Lemma B.82 (Interaction of environment subsumption and length) *If* $\Phi \subseteq \Phi'$ *then* $|\Phi| \leq |\Phi'|$.

By induction on $\Phi \subseteq \Phi'$.

Lemma B.83 (Interaction of environment subsumption and extensions substitution) *If* $\Phi \subseteq \Phi'$ *then* $\Phi \cdot \sigma_\Psi \subseteq \Phi' \cdot \sigma_\Psi$.

By induction on $\Phi \subseteq \Phi'$.

Lemma B.84 (Interaction of extensions substitution and element access) *1.* $(\sigma.\mathbf{I}) \cdot \sigma_\Psi = (\sigma \cdot \sigma_\Psi).\mathbf{I} \cdot \sigma_\Psi$
2. $(\Phi.\mathbf{I}) \cdot \sigma_\Psi = (\Phi \cdot \sigma_\Psi).\mathbf{I} \cdot \sigma_\Psi$

By induction on $\mathbf{I}$ and taking into account the implicit assumption that $\mathbf{I} < |\sigma|$ or $\mathbf{I} < |\Phi|$.

Lemma B.85 (Extension of lemma B.30) *If* $\Psi \vdash \sigma_\Psi : \Psi'$ *and* $\sigma_\Psi.i = [\Phi]\,t$ *then* $t <^f |\Phi|$ *and* $t <^b 0$.

Identical as before.

Lemma B.86 (Extension of lemma B.31) *If* $t <^b n$ *then* $\lceil t \cdot \sigma \rceil_m^n = t \cdot \lceil \sigma \rceil_m^n$.

Identical as before.

Lemma B.87 (Extension of lemma B.32) *1. If* $\Psi \vdash \sigma_\Psi : \Psi'$ *then* $\lceil t \rceil_{\mathbf{I}}^n \cdot \sigma_\Psi = \lceil t \cdot \sigma_\Psi \rceil_{\mathbf{I} \cdot \sigma_\Psi}^n$
2. If $\Psi \vdash \sigma_\Psi : \Psi'$ *then* $\lceil \sigma \rceil_{\mathbf{I}}^n \cdot \sigma_\Psi = \lceil \sigma \cdot \sigma_\Psi \rceil_{\mathbf{I} \cdot \sigma_\Psi}^n$

Part 1 is proved by induction on t.
In the case $t = b_n$, we have that the left-hand side is equal to $f_{\mathbf{I}} \cdot \sigma_\Psi = f_{\mathbf{I}\cdot\sigma_\Psi}$. The right-hand side is equal to $\lceil b_n \rceil^n_{\mathbf{I}\cdot\sigma_\Psi} = f_{\mathbf{I}\cdot\sigma_\Psi}$.
In the case $t = X_i/\sigma$, this is proved entirely as before, with trivial changes to account for the new indexes.
Part 2 is proved by induction on σ, as previously. For the new case $\sigma = \sigma'$, $\mathbf{id}(X_i)$, the result is trivial.

Lemma B.88 (Extension of lemma B.33) *If $t <^b n$ then $\lfloor t \cdot \sigma \rfloor^n_{\mathbf{I}} = t \cdot \lfloor \sigma \rfloor^n_{\mathbf{I}}$.*

Identical as before.

Lemma B.89 (Extension of lemma B.34) *1. If $\Psi \vdash \sigma_\Psi : \Psi'$ then $\lfloor t \rfloor^n_{\mathbf{I}} \cdot \sigma_\Psi = \lfloor t \cdot \sigma_\Psi \rfloor^n_{\mathbf{I}\cdot\sigma_\Psi}$*
2. If $\Psi \vdash \sigma_\Psi : \Psi'$ then $\lfloor \sigma \rfloor^n_{\mathbf{I}} \cdot \sigma_\Psi = \lfloor \sigma \cdot \sigma_\Psi \rfloor^n_{\mathbf{I}\cdot\sigma_\Psi}$

Proved similarly to lemma B.87.
When $t = f_{\mathbf{I}}$, we have that the left-hand side is equal to b_n, while the right-hand side is equal to $\lfloor f_{\mathbf{I}\cdot\sigma_\Psi} \rfloor^n_{\mathbf{I}\cdot\sigma_\Psi} = b_n$.

Lemma B.90 (Extension of lemma B.35) *1. $(t \cdot \sigma) \cdot \sigma_\Psi = (t \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$*
2. $(\sigma \cdot \sigma') \cdot \sigma_\Psi = (\sigma \cdot \sigma_\Psi) \cdot (\sigma' \cdot \sigma_\Psi)$

Part 1 is entirely similar as before, with the exception of case $t = f_{\mathbf{I}}$. This is proved using the lemma B.84. Part 2 is trivially proved for the new case of σ.

Lemma B.91 (Extension of lemma B.36) *$id_\Phi \cdot \sigma_\Psi = id_{\Phi\cdot\sigma_\Psi}$*

By induction on Φ.
When $\Phi = \bullet$, trivial.
When $\Phi = \Phi'$, t, by induction we have $\mathsf{id}_{\Phi'} \cdot \sigma_\Psi = \mathsf{id}_{\Phi'\cdot\sigma_\Psi}$. Thus $(\mathsf{id}_{\Phi'},\ f_{|\Phi'|}) \cdot \sigma_\Psi = \mathsf{id}_{\Phi'\cdot\sigma_\Psi},\ f_{|\Phi'|\cdot\sigma_\Psi} = \mathsf{id}_{\Phi'\cdot\sigma_\Psi},\ f_{|\Phi'\cdot\sigma_\Psi|} = \mathsf{id}_{\Phi\cdot\sigma_\Psi}$.
When $\Phi = \Phi'$, X_i, we have that $\mathsf{id}_{\Phi'\cdot\sigma_\Psi}, \mathsf{id}_{\sigma_\Psi.i} = \mathsf{id}_{\Phi'\cdot\sigma_\Psi,\ \sigma_\Psi.i}$ (by simple induction on $\Phi'' = \sigma_\Psi.i$).

Lemma B.92 (Extension of lemma B.37) *1. If $\Psi;\ \Phi \vdash t : t'$, $|\sigma_\Psi| = |\Psi|$ and $\sigma_\Psi \subseteq \sigma'_\Psi$ then $t \cdot \sigma'_\Psi = t \cdot \sigma_\Psi$.*
2. If $\Psi;\ \Phi \vdash \sigma : \Phi'$, $|\sigma_\Psi| = |\Psi|$ and $\sigma_\Psi \subseteq \sigma'_\Psi$ then $\sigma \cdot \sigma'_\Psi = \sigma \cdot \sigma_\Psi$.
3. If $\Psi \vdash \Phi$ wf, $|\sigma_\Psi| = |\Psi|$ and $\sigma_\Psi \subseteq \sigma'_\Psi$ then $\Phi \cdot \sigma'_\Psi = \Phi \cdot \sigma_\Psi$.
4. If $\Psi \vdash T : K$, $|\sigma_\Psi| = |\Psi|$ and $\sigma_\Psi \subseteq \sigma'_\Psi$ then $T \cdot \sigma'_\Psi = T \cdot \sigma_\Psi$.
5. If $K \cdot \sigma_\Psi$ is well-defined, and $\sigma_\Psi \subseteq \sigma'_\Psi$, then $K \cdot \sigma_\Psi = K \cdot \sigma'_\Psi$.
6. If $\Psi \cdot \sigma_\Psi$ is well-defined, and $\sigma_\Psi \subseteq \sigma'_\Psi$, then $\Psi \cdot \sigma_\Psi = \Psi \cdot \sigma'_\Psi$.

Parts 2 and 3 are trivially extended for the new cases; others are identical or easily provable by induction.

Lemma B.93 (Extension of lemma B.38) *If $\vdash \Psi$ wf and $\Psi \vdash \sigma_\Psi : \Psi'$ then $\Psi \vdash \sigma_\Psi.i : \Psi'.i \cdot \sigma_\Psi$.*

By induction on σ_Ψ and then cases on $i < |\sigma_\Psi|$.
If $i = |\sigma_\Psi| - 1$ then proceed by cases for σ_Ψ.
If $\sigma_\Psi = \bullet$, then the case is impossible.
If $\sigma_\Psi = \sigma'_\Psi,\ [\Phi]t$, we have by typing inversion for σ_Ψ that $\Psi \vdash [\Phi]t : (\Psi'.i) \cdot \sigma'_\Psi$, which by lemma B.92 is equal to the desired.
If $\sigma_\Psi = \sigma'_\Psi,\ [\Phi]\Phi'$, we get by typing inversion for σ_Ψ that $\Psi \vdash [\Phi]\Phi' : [\Psi'.i \cdot \sigma'_\Psi]\,\mathsf{ctx}$ which again by lemma B.92 is the desired.
If $i < |\sigma_\Psi| - 1$ then by inversion of σ_Ψ we have that either $\sigma_\Psi = \sigma'_\Psi,\ [\Phi]t$ or $\sigma_\Psi = \sigma'_\Psi,\ [\Phi]\Phi'$. In both cases $i < |\sigma'_\Psi| - 1$ so by induction hypothesis get $\sigma'_\Psi.i : \Psi'.i \cdot \sigma'_\Psi$ which, using B.92, is the desired.

Lemma B.94 (Interaction of two extension substitutions) *1.* $(\mathbf{I}\cdot\sigma_\Psi)\cdot\sigma'_\Psi = \mathbf{I}\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

2. $(t\cdot\sigma_\Psi)\cdot\sigma'_\Psi = t\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

3. $(\Phi\cdot\sigma_\Psi)\cdot\sigma'_\Psi = \Phi\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

4. $(\sigma\cdot\sigma_\Psi)\cdot\sigma'_\Psi = \sigma\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

5. $(T\cdot\sigma_\Psi)\cdot\sigma'_\Psi = T\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

6. $(K\cdot\sigma_\Psi)\cdot\sigma'_\Psi = K\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

7. $(\Psi\cdot\sigma_\Psi)\cdot\sigma'_\Psi = \Psi\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$

Part 1 By induction on $\mathbf{I}$. The interesting case is $\mathbf{I} = \mathbf{I}'$, X_i. In that case we have $(\mathbf{I}\cdot\sigma_\Psi)\cdot\sigma'_\Psi = (\mathbf{I}'\cdot\sigma_\Psi)\cdot\sigma'_\Psi,\ \sigma_\Psi.i\cdot\sigma'_\Psi$. Trivially $\sigma_\Psi.i\cdot\sigma'_\Psi = (\sigma_\Psi\cdot\sigma'_\Psi).i$, and also using induction hypothesis, we have that the above is further equal to $\mathbf{I}'\cdot(\sigma_\Psi\cdot\sigma'_\Psi),\ (\sigma_\Psi\cdot\sigma'_\Psi).i$, which is exactly the desired.

Part 2 By induction on t. The interesting case is $t = X_i/\sigma$. The left-hand-side is then equal to $(\sigma_\Psi.i\cdot(\sigma\cdot\sigma_\Psi))\cdot\sigma'_\Psi$, with $\sigma_\Psi.i = [\Phi]t$. This is further rewritten as $(t\cdot(\sigma\cdot\sigma_\Psi))\cdot\sigma'_\Psi = (t\cdot\sigma'_\Psi)\cdot((\sigma\cdot\sigma_\Psi)\cdot\sigma'_\Psi)$ through lemma B.90. Furthermore through part 4 we get that this is equal to $(t\cdot\sigma'_\Psi)\cdot(\sigma\cdot(\sigma_\Psi\cdot\sigma'_\Psi))$.
The right-hand-side is written as: $(X_i/\sigma)\cdot(\sigma_\Psi\cdot\sigma'_\Psi)$. We have that $(\sigma_\Psi\cdot\sigma'_\Psi).i = (\sigma_\Psi.i)\cdot\sigma'_\Psi = [\Phi\cdot\sigma'_\Psi](t\cdot\sigma'_\Psi)$. Thus $(X_i/\sigma)\cdot(\sigma_\Psi\cdot\sigma'_\Psi) = (t\cdot\sigma'_\Psi)\cdot(\sigma\cdot(\sigma_\Psi\cdot\sigma'_\Psi))$.

Part 3 By induction on Φ. When $\Phi = \Phi,\ X_i$, we have that the left-hand-side is equal to $(\Phi\cdot\sigma_\Psi)\cdot\sigma'_\Psi,\ \Phi'\cdot\sigma'_\Psi$ with $\sigma_\Psi.i = [\Phi]\Phi'$. By induction hypothesis this is further equal to $\Phi\cdot(\sigma_\Psi\cdot\sigma'_\Psi),\ \Phi'\cdot\sigma'_\Psi$.
Also, we have that $(\sigma_\Psi\cdot\sigma'_\Psi).i = [\Phi\cdot\sigma'_\Psi]\,\Phi'\cdot\sigma'_\Psi$. Thus the right-hand-side is equal to $\Phi\cdot(\sigma_\Psi\cdot\sigma'_\Psi),\ \Phi'\cdot\sigma'_\Psi$, which is exactly equal to the left-hand-side.

Rest Similarly as above.

Lemma B.95 (Interaction of identity substitution and extension substitution) *If* $|\sigma_\Psi| = |\Psi|$ *then* $id_\Psi\cdot\sigma_\Psi = \sigma_\Psi$

By induction on Ψ. If $\Psi = \bullet$, trivial. If $\Psi = \Psi',\ K$ then $\mathsf{id}_{\Psi',K}\cdot\sigma_\Psi = (\mathsf{id}_{\Psi'},\ X_{|\Psi'|})\cdot\sigma_\Psi$. From $|\sigma_\Psi| = |\Psi|$ we have that $\sigma_\Psi = \sigma'_\Psi,\ T$, and from induction hypothesis for σ'_Ψ we get that the above is equal to $\sigma'_\Psi,\ X_{|\Psi'|}\cdot\sigma_\Psi = \sigma'_\Psi,\ T = \sigma_\Psi$.

Part 2

Lemma B.96 (Interaction of identity substitution and extension substitution) *1.* $t\cdot id_\Psi = t$

2. $\Phi\cdot id_\Psi = \Phi$

3. $\sigma\cdot id_\Psi = \sigma$

4. $T\cdot id_\Psi = T$

5. $K\cdot id_\Psi = K$

6. $\sigma_\Psi\cdot id_\Psi = \sigma_\Psi$

All are trivially proved by induction. We will give only details for the σ_Ψ case.
By induction on σ_Ψ. If $\sigma_\Psi = \bullet$, trivial. If $\sigma_\Psi = \sigma'_\Psi,\ T$, then we have that $(\sigma'_\Psi, T)\cdot\mathsf{id}_\Psi = \sigma'_\Psi\cdot\mathsf{id}_\Psi,\ T\cdot\mathsf{id}_\Psi$. The first part is equal to σ'_Ψ by induction hypothesis (and use of lemma B.92). For the second we split cases for T. We have $([\Phi]t)\cdot\mathsf{id}_\Psi = [\Phi\cdot\mathsf{id}_\Psi](t\cdot\mathsf{id}_\Psi) = [\Phi]t$, and similarly for $([\Phi]\Phi')\cdot\mathsf{id}_\Psi = [\Phi]\Phi'$, by use of the other parts.

Theorem B.97 (Extension of lemma B.39) *1. If* $\Psi;\ \Phi\vdash t : t'$ *and* $\Psi'\vdash\sigma_\Psi : \Psi$ *then* $\Psi';\ \Phi\cdot\sigma_\Psi\vdash t\cdot\sigma_\Psi : t'\cdot\sigma_\Psi$.

2. *If* $\Psi;\ \Phi \vdash \sigma : \Phi'$ *and* $\Psi' \vdash \sigma_\Psi : \Psi$ *then* $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$.

3. *If* $\Psi \vdash \Phi$ *wf and* $\Psi' \vdash \sigma_\Psi : \Psi$ *then* $\Psi' \vdash \Phi \cdot \sigma_\Psi$ *wf.*

4. *If* $\Psi \vdash T : K$ *and* $\Psi' \vdash \sigma_\Psi : \Psi$ *then* $\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi$.

5. *If* $\Psi' \vdash \sigma_\Psi : \Psi$ *and* $\Psi'' \vdash \sigma'_\Psi : \Psi'$ *then* $\Psi'' \vdash \sigma_\Psi \cdot \sigma'_\Psi : \Psi$.

Part 1. **Case** $\dfrac{\Phi.\mathbf{I} = t}{\Psi;\ \Phi \vdash f_{\mathbf{I}} : t}\ \triangleright$

We have $(\Phi \cdot \sigma_\Psi).\mathbf{I} \cdot \sigma_\Psi = (\Phi.\mathbf{I}) \cdot \sigma_\Psi$ from lemma B.84.

Case $\dfrac{\Psi.i = T \qquad T = [\Phi']\, t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\ \triangleright$

From lemma B.93 get $\Psi' \vdash \sigma_\Psi.i : (\Psi.i) \cdot \sigma_\Psi$.
Furthermore, this can be written as:
$\Psi' \vdash \sigma_\Psi.i : [\Phi' \cdot \sigma_\Psi]\, t' \cdot \sigma_\Psi$.
Thus by typing inversion, and assuming $\sigma_\Psi.i = [\Phi' \cdot \sigma_\Psi]\, t$ get:
$\Psi';\ \Phi' \cdot \sigma_\Psi \vdash t : t' \cdot \sigma_\Psi$. From part 2 for σ get $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$.
From lemma B.67 and the above we get $\Psi';\ \Phi \cdot \sigma_\Psi \vdash t \cdot (\sigma \cdot \sigma_\Psi) : (t' \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi)$.
Using the lemma B.90 we get that $(t' \cdot \sigma_\Psi) \cdot (\sigma \cdot \sigma_\Psi) = (t' \cdot \sigma) \cdot \sigma_\Psi$, thus the above is the desired.

Case (otherwise) $\triangleright$
The rest of the cases are trivial to adapt to account for indexes from lemma B.39.

Part 2. The cases for $\sigma = \bullet$ or $\sigma = \sigma',\ t$ are entirely similar as before.

Case $\dfrac{\Psi;\ \Phi \vdash \sigma : \Phi' \qquad \Psi.i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\ X_i \subseteq \Phi}{\Psi;\ \Phi \vdash (\sigma,\ \mathbf{id}(X_i)) : (\Phi',\ X_i)}\ \triangleright$

In this case we need to prove that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash (\sigma \cdot \sigma_\Psi,\ \mathsf{id}_{\sigma_\Psi.i}) : (\Phi' \cdot \sigma_\Psi,\ \sigma_\Psi.i)$.
By induction hypothesis for σ we get that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$.
From lemma B.93 we also get: $\Psi' \vdash \sigma_\Psi.i : \Psi.i \cdot \sigma_\Psi$.
We have that $\Psi.i = [\Phi']\,\mathsf{ctx}$, so this can be rewritten as: $\Psi' \vdash \sigma_\Psi.i : [\Phi' \cdot \sigma_\Psi]\,\mathsf{ctx}$.
By typing inversion get $\sigma_\Psi.i = [\Phi' \cdot \sigma_\Psi]\,\Phi''$ for some Φ'' and:
$\Psi' \vdash [\Phi' \cdot \sigma_\Psi]\,\Phi'' : [\Phi' \cdot \sigma_\Psi]\,\mathsf{ctx}$.
Now proceed by induction on Φ'' to prove that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash (\sigma \cdot \sigma_\Psi,\ \mathsf{id}_{\sigma_\Psi.i}) : (\Phi' \cdot \sigma_\Psi,\ \sigma_\Psi.i)$.
When $\Phi'' = \bullet$, trivial.
When $\Phi'' = \Phi''',\ t$, have $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi,\ \mathsf{id}_{[\Phi' \cdot \sigma_\Psi]\,\Phi'''} : (\Phi' \cdot \sigma_\Psi,\ \Phi''')$ by induction hypothesis. We can append $f_{|\Phi' \cdot \sigma_\Psi|,|\Phi'''|}$ to this substitution and get the desired, because $(|\Phi' \cdot \sigma_\Psi|, |\Phi'''|) < |\Phi \cdot \sigma_\Psi|$. This is because $(\Phi',\ X_i) \subseteq \Phi$ thus $(\Phi' \cdot \sigma_\Psi,\ \Phi''',\ t) \subseteq \Phi$ and thus $(|\Phi' \cdot \sigma_\Psi|,\ |\Phi'''|, \cdot) \leq |\Phi|$. When $\Phi'' = \Phi''',\ X_j$, have $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi,\ \mathsf{id}_{[\Phi' \cdot \sigma_\Psi]\,\Phi'''} : (\Phi' \cdot \sigma_\Psi,\ \Phi''')$. Now we have that $\Phi',\ X_i \subseteq \Phi$, which also means that $(\Phi' \cdot \sigma_\Psi,\ \Phi''',\ X_j) \subseteq \Phi \cdot \sigma_\Psi$. Thus we can apply the typing rule for $\mathbf{id}(X_j)$ to get that $\Psi';\ \Phi \cdot \sigma_\Psi \vdash \sigma \cdot \sigma_\Psi,\ \mathsf{id}_{[\Phi' \cdot \sigma_\Psi]\,\Phi'''},\ \mathbf{id}(X_j) : (\Phi' \cdot \sigma_\Psi,\ \Phi''',\ X_j)$, which is the desired.

Part 3. **Case** $\dfrac{}{\Psi \vdash \bullet\ \mathrm{wf}}\ \triangleright$

Trivial.

Case $\dfrac{\Psi \vdash \Phi \text{ wf} \qquad \Psi;\, \Phi \vdash t : s}{\Psi \vdash (\Phi,\, t) \text{ wf}}$ $\triangleright$
By induction hypothesis we get $\Psi' \vdash \Phi \cdot \Psi$ wf.
By use of part 1 we get that $\Psi';\, \Phi \cdot \Psi \vdash t \cdot \Psi : s$.
Thus using the same typing rule we get the desired $\Psi' \vdash (\Phi \cdot \Psi,\, t \cdot \Psi)$ wf.

Case $\dfrac{\Psi \vdash \Phi \text{ wf} \qquad \Psi.i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash (\Phi,\, X_i) \text{ wf}}$ $\triangleright$
By induction hypothesis we get $\Psi' \vdash \Phi \cdot \sigma_\Psi$ wf.
By use of lemma B.93 we get that $\Psi' \vdash \sigma_\Psi.i : \Psi.i \cdot \sigma_\Psi$.
We have $\Psi.i = [\Phi]\,\mathsf{ctx}$ thus the above can be rewritten as $\Psi' \vdash \sigma_\Psi.i : [\Phi \cdot \sigma_\Psi]\,\mathsf{ctx}$.
By inversion of typing get that $\sigma_\Psi.i = [\Phi \cdot \sigma_\Psi]\,\Phi'$ and that $\Psi' \vdash \Phi \cdot \sigma_\Psi,\, \Phi'$ wf. This is exactly the desired result.

Part 4. **Case** $\dfrac{\Psi;\, \Phi \vdash t : t'}{\Psi \vdash [\Phi]\,t : [\Phi]\,t'}$ $\triangleright$
By use of part 1 we get that $\Psi';\, \Phi \cdot \sigma_\Psi \vdash t \cdot \sigma_\Psi : t' \cdot \sigma_\Psi$.
Thus by application of the same typing rule we get exactly the desired.

Case $\dfrac{\Psi \vdash \Phi,\, \Phi' \text{ wf}}{\Psi \vdash [\Phi]\,\Phi' : [\Phi]\,\mathsf{ctx}}$ $\triangleright$
By use of part 3 we get $\Psi' \vdash \Phi \cdot \sigma_\Psi,\, \Phi' \cdot \sigma_\Psi$ wf.
Thus by the same typing rule we get exactly the desired.

Part 5. **Case** $\dfrac{}{\Psi' \vdash \bullet : \bullet}$ $\triangleright$
Trivial.

Case $\dfrac{\Psi' \vdash \sigma_\Psi : \Psi \qquad \Psi' \vdash T : K \cdot \sigma_\Psi}{\Psi' \vdash (\sigma_\Psi,\, T) : (\Psi,\, K)}$ $\triangleright$
By induction we get $\Psi'' \vdash \sigma_\Psi \cdot \sigma'_\Psi : \Psi$.
By use of part 4 we get $\Psi'' \vdash T \cdot \sigma'_\Psi : (K \cdot \sigma_\Psi) \cdot \sigma'_\Psi$.
This is equal to $K \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$ by use of lemma B.94. Thus we get the desired result by applying the same typing rule.

Lemma B.98 *If* $\Psi \vdash \Psi''$ *wf and* $\Psi' \vdash \sigma_\Psi : \Psi$ *then* $\Psi' \vdash \Psi'' \cdot \sigma_\Psi$ *wf.*

By induction on the structure of Ψ''.

Case $\Psi'' = \bullet$ $\triangleright$ Trivial.

Case $\Psi'' = \Psi'',\, [\Phi]\,t$ $\triangleright$
By induction hypothesis we have that $\Psi' \vdash \Psi'' \cdot \sigma_\Psi$ wf.
By inversion of well-formedness for $\Psi'',\, [\Phi]\,t$ we get:
$\Psi,\, \Psi'' \vdash [\Phi]\,t : [\Phi]\,s$.
We have for $\sigma'_\Psi = \sigma_\Psi,\, X_{|\Psi'|},\, \cdots,\, X_{|\Psi'|+|\Psi''|}$, that $\Psi',\, \Psi'' \cdot \sigma_\Psi \vdash \sigma'_\Psi : \Psi,\, \Psi''$.
Thus by application of lemma B.97, we get that:

Ψ', $\Psi'' \cdot \sigma_\Psi \vdash [\Phi \cdot \sigma'_\Psi]\, t \cdot \sigma'_\Psi : [\Phi \cdot \sigma'_\Psi]\, s$.
Thus $\vdash \Psi'$, $(\Psi'', [\Phi]\, t) \cdot \sigma_\Psi$ wf, which is the desired.

Case $\Psi'' = \Psi''$, $[\Phi]\,\mathsf{ctx} \triangleright$ Similarly as the previous case.

B.3 Final extension: bound extension variables

The metatheory presented in the previous subsection only has to do with meta and context variables that are free. We now introduce bound extension variables, (which will be bound in the computational language), entirely similarly to how we have bound and free variables for the logic. We will not re-prove everything here; all theorems from above carry on exactly as they are. We will only prove two theorems that have to do with the interaction of freshen/bind and extension substitutions.

Definition B.99 (Syntax of the language) *The syntax of the logic language is extended below.*

$$
\begin{aligned}
\Phi &::= \cdots \mid \Phi, B_i \\
\sigma &::= \cdots \mid \sigma,\ \mathbf{id}(B_i) \\
t &::= \cdots \mid B_i/\sigma \\
\mathbf{I} &::= \cdots \mid \mathbf{I},\ |B_i|
\end{aligned}
$$

All the following definitions are extended trivially. Application of extension substitution leaves bound extension variables as they are. Bound extension variables are untypable.

Definition B.100 (Freshening of extension variables) *We define freshening similarly to normal variables. We do not define extension variables limits: we will use the condition of well-definedness later. (So if $\lceil t \rceil^M_{N,K}$ is well-defined, that means that it does not have extension variables larger than $N+K$).*

$\boxed{\lceil \mathbf{I} \rceil^M_N}$

$$
\begin{aligned}
\lceil \bullet \rceil &= \bullet \\
\lceil \mathbf{I}, \cdot \rceil &= \lceil \mathbf{I} \rceil, \cdot \\
\lceil \mathbf{I}, X_i \rceil &= \lceil \mathbf{I} \rceil, X_i \\
\lceil \mathbf{I}, B_{M+j} \rceil^M_{N,K} &= \lceil \mathbf{I} \rceil, X_{N+K-j-1} \text{ when } j < K \\
\lceil \mathbf{I}, B_i \rceil^M_{N,K} &= \lceil \mathbf{I} \rceil, B_i \text{ when } i < M
\end{aligned}
$$

$\boxed{\lceil t \rceil^M_{N,K}}$

$$
\begin{aligned}
\lceil f_{\mathbf{I}} \rceil^M_{N,K} &= f_{\lceil \mathbf{I} \rceil^M_{N,K}} \\
\lceil b_i \rceil^M_{N,K} &= b_i \\
\lceil \lambda(t_1).t_2 \rceil^M_{N,K} &= \lambda(\lceil t_1 \rceil^M_{N,K}).\, \lceil t_2 \rceil^M_{N,K} \\
&\cdots \\
\lceil X_i/\sigma \rceil^M_{N,K} &= X_i/(\lceil \sigma \rceil^M_{N,K}) \\
\lceil B_{M+j}/\sigma \rceil^M_{N,K} &= X_{N+K-j-1}/(\lceil \sigma \rceil^M_{N,K}) \text{ when } j < K \\
\lceil B_i/\sigma \rceil^M_{N,K} &= B_i/(\lceil \sigma \rceil^M_{N,K}) \text{ when } i < M
\end{aligned}
$$

$\boxed{\lceil \Phi \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil \bullet \rceil & = & \bullet \\
\lceil \Phi, t \rceil & = & \lceil \Phi \rceil, \lceil t \rceil \\
\lceil \Phi, X_i \rceil & = & \lceil \Phi \rceil, X_i \\
\lceil \Phi, B_{M+j} \rceil^M_{N,K} & = & \lceil \Phi \rceil, X_{N+K-j-1} \text{ when } j < K \\
\lceil \Phi, B_i \rceil^M_{N,K} & = & \lceil \Phi \rceil, B_i \text{ when } i < M
\end{array}
$$

$\boxed{\lceil \sigma \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil \bullet \rceil & = & \bullet \\
\lceil \sigma, t \rceil & = & \lceil \sigma \rceil, \lceil t \rceil \\
\lceil \sigma, \mathbf{id}(X_i) \rceil & = & \lceil \sigma \rceil, \mathbf{id}(X_i) \\
\lceil \sigma, \mathbf{id}(B_{M+j}) \rceil^M_{N,K} & = & \lceil \sigma \rceil, \mathbf{id}(X_{N+K-j-1}) \text{ when } j < K \\
\lceil \sigma, \mathbf{id}(B_i) \rceil^M_{N,K} & = & \lceil \sigma \rceil, \mathbf{id}(B_i) \text{ when } i < M
\end{array}
$$

$\boxed{\lceil T \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil [\Phi] t \rceil & = & [\lceil \Phi \rceil](\lceil t \rceil) \\
\lceil [\Phi] \Phi' \rceil & = & [\lceil \Phi \rceil](\lceil \Phi' \rceil)
\end{array}
$$

$\boxed{\lceil K \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil [\Phi] t \rceil & = & [\lceil \Phi \rceil](\lceil t \rceil) \\
\lceil [\Phi] \mathsf{ctx} \rceil & = & [\lceil \Phi \rceil] \mathsf{ctx}
\end{array}
$$

$\boxed{\lceil \Psi \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil \bullet \rceil^M_{N,K} & = & \bullet \\
\lceil \Psi,\, K \rceil^M_{N,K} & = & \lceil \Psi \rceil^M_{N,K}, \lceil K \rceil^{M+|\Psi|}_{N,K}
\end{array}
$$

Definition B.101 (Binding of extension variables) *We define binding similarly to normal variables. Note that this is a bit different (because binding many variables at once is permitted), so the N parameter is the length of the resulting context (the number of free variables after binding has taken place), while $N+K$ is the length of the context where the bind argument is currently in.*

$\boxed{\lfloor \mathbf{I} \rfloor^M_{N,K}}$

$$
\begin{array}{lcl}
\lfloor \bullet \rfloor & = & \bullet \\
\lfloor \mathbf{I}, \cdot \rfloor & = & \lfloor \mathbf{I} \rfloor, \cdot \\
\lfloor \mathbf{I}, X_{N+j} \rfloor^M_{N,K} & = & \lfloor \mathbf{I} \rfloor, B_{M+K-j-1} \text{ when } j < K \\
\lfloor \mathbf{I}, X_i \rfloor^M_{N,K} & = & \lfloor \mathbf{I} \rfloor, X_i \text{ when } i < N \\
\lfloor \mathbf{I}, B_i \rfloor^M_{N,K} & = & \lfloor \mathbf{I} \rfloor, B_i
\end{array}
$$

$\boxed{\lfloor t \rfloor^{M}_{N,K}}$

$$
\begin{array}{lcl}
\lfloor f_{\mathbf{I}} \rfloor^{M}_{N,K} & = & f_{\lfloor \mathbf{I} \rfloor^{M}_{N,K}} \\
\lfloor b_i \rfloor^{M}_{N,K} & = & b_i \\
\lfloor \lambda(t_1).t_2 \rfloor^{M}_{N,K} & = & \lambda(\lfloor t_1 \rfloor^{M}_{N,K}).\, \lfloor t_2 \rfloor^{M}_{N,K} \\
 & & \cdots \\
\lfloor X_{N+j}/\sigma \rfloor^{M}_{N,K} & = & B_{M+K-j-1}/(\lfloor \sigma \rfloor^{M}_{N,K}) \text{ when } j < K \\
\lfloor X_i/\sigma \rfloor^{M}_{N,K} & = & X_i/(\lfloor \sigma \rfloor^{M}_{N,K}) \text{ when } i < N \\
\lfloor B_i/\sigma \rfloor^{M}_{N,K} & = & B_i/(\lfloor \sigma \rfloor^{M}_{N,K})
\end{array}
$$

$\boxed{\lfloor \Phi \rfloor^{M}_{N,K}}$

$$
\begin{array}{lcl}
\lfloor \bullet \rfloor & = & \bullet \\
\lfloor \Phi, t \rfloor & = & \lfloor \Phi \rfloor,\, \lfloor t \rfloor \\
\lfloor \Phi, X_{N+j} \rfloor^{M}_{N,K} & = & \lfloor \Phi \rfloor,\, B_M \text{ when } j < K \\
\lfloor \Phi, X_i \rfloor^{M}_{N,K} & = & \lfloor \Phi \rfloor,\, X_i \text{ when } i < N \\
\lfloor \Phi, B_i \rfloor^{M}_{N,K} & = & \lfloor \Phi \rfloor,\, B_i
\end{array}
$$

$\boxed{\lfloor \sigma \rfloor^{M}_{N,K}}$

$$
\begin{array}{lcl}
\lfloor \bullet \rfloor & = & \bullet \\
\lfloor \sigma, t \rfloor & = & \lfloor \sigma \rfloor,\, \lfloor t \rfloor \\
\lfloor \sigma, \mathbf{id}(X_{N+j}) \rfloor^{M}_{N,K} & = & \lfloor \sigma \rfloor,\, \mathbf{id}(B_{M+K-j-1}) \text{ when } j < K \\
\lfloor \sigma, \mathbf{id}(X_i) \rfloor^{M}_{N,K} & = & \lfloor \sigma \rfloor,\, \mathbf{id}(X_i) \text{ when } i < N \\
\lfloor \sigma, \mathbf{id}(B_i) \rfloor^{M}_{N,K} & = & \lfloor \sigma \rfloor,\, \mathbf{id}(B_i)
\end{array}
$$

$\boxed{\lfloor T \rfloor^{M}_{N,K}}$

$$
\begin{array}{lcl}
\lfloor [\Phi] t \rfloor & = & [\lfloor \Phi \rfloor](\lfloor t \rfloor) \\
\lfloor [\Phi] \Phi' \rfloor & = & [\lfloor \Phi \rfloor](\lfloor \Phi' \rfloor)
\end{array}
$$

$\boxed{\lfloor K \rfloor^{M}_{N,K}}$

$$
\begin{array}{lcl}
\lfloor [\Phi] t \rfloor & = & [\lfloor \Phi \rfloor](\lfloor t \rfloor) \\
\lfloor [\Phi] \mathsf{ctx} \rfloor & = & [\lfloor \Phi \rfloor] \mathsf{ctx}
\end{array}
$$

$\boxed{\lfloor \Psi \rfloor^{M}_{N,K}}$

$$
\begin{array}{lcl}
\lfloor \bullet \rfloor^{M}_{N,K} & = & \bullet \\
\lfloor \Psi,\, K \rfloor^{M}_{N,K} & = & \lfloor \Psi \rfloor^{M}_{N,K},\, \lfloor K \rfloor^{M+|\Psi|}_{N,K}
\end{array}
$$

Definition B.102 *Opening up and closing down an extension context works as follows:*

$\boxed{\uparrow\Psi\uparrow_N}$

$$\begin{array}{lcl} \uparrow\bullet\uparrow_N & = & \bullet \\ \uparrow\Psi,\, K\uparrow_N & = & \uparrow\Psi\uparrow_N,\, \uparrow K\uparrow^0_{N,|\Psi|} \end{array}$$

$\boxed{\downarrow\Psi\downarrow_N}$

$$\begin{array}{lcl} \downarrow\bullet\downarrow_N & = & \bullet \\ \downarrow\Psi,\, K\downarrow_N & = & \downarrow\Psi\downarrow_N,\, \downarrow K\downarrow^0_{N,|\Psi|} \end{array}$$

Now we prove a couple of theorems.

Lemma B.103 (Freshening of extension variables and extension substitution) *Assuming* $|\sigma_\Psi| = N$, $\lceil\cdot\rceil^M_{N,K}$ *and* $\lceil\cdot\rceil^M_{N',K}$ *are well-defined, we have:*

1. $\lceil \mathbf{I}\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil\mathbf{I}\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$
2. $\lceil t\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil t\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$
3. $\lceil \Phi\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil\Phi\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$
4. $\lceil \sigma\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil\sigma\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$
5. $\lceil T\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil T\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$
6. $\lceil K\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil K\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$
7. $\lceil \Psi\cdot\sigma_\Psi\rceil^M_{N',K} = \lceil\Psi\rceil^M_{N,K}\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1})$

Part 2 By induction on t and use of the rest of the parts. The interesting case is $t = B_{M+j}/sigma$ with $j < K$. We have that the left-hand-side is equal to $X_{N'+K-j-1}/(\lceil\sigma\cdot\sigma_\Psi\rceil^M_{N',K})$, which by part 4 is equal to $X_{N'+K-j-1}/(\lceil\sigma\rceil^M_{N,K}\cdot(\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$. The right-hand-side is equal to $(X_{N+K-j-1}/\lceil\sigma\rceil^M_{N,K})\cdot(\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1}) = X_{N'+K-j-1}/(\lceil\sigma\rceil^M_{N,K}\cdot(\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$, which is exactly equal to the left-hand-side.

Part 7 By induction on Ψ. The interesting case occurs when $\Psi = \Psi',\, K$.
In that case, we have that the left-hand-side is equal to:
$\left\lceil\Psi'\cdot\sigma_\Psi,\, K\cdot(\sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+|\Psi'|})\right\rceil^M_{N',K}$.
Since K does not contain variables bigger than $X_{|\sigma_\Psi|}$ (since $\lceil K\rceil^M_{N,K}$ is well-defined), we have that this is further equal to:
$\lceil\Psi'\cdot\sigma_\Psi,\, K\cdot\sigma_\Psi\rceil^M_{N',K}$.
This is then equal to:
$\lceil\Psi'\cdot\sigma_\Psi\rceil^M_{N',K},\, \lceil K\cdot\sigma_\Psi\rceil^{M+|\Psi'\cdot\sigma_\Psi|}_{N',K}$. Setting $\sigma'_\Psi = \sigma_\Psi,\, X_{N'},\, \cdots,\, X_{N'+K-1}$ we have by induction hypothesis and part 6 that this is equal to:
$\lceil\Psi'\rceil^M_{N,K}\cdot\sigma'_\Psi,\, \lceil K\rceil^{M+|\Psi'\cdot\sigma_\Psi|}_{N,K}\cdot\sigma'_\Psi$.
The right-hand-side is equal to: $\lceil\Psi'\rceil^M_{N,K}\cdot\sigma'_\Psi,\, \lceil K\rceil^{M+|\Psi'|}_{N,K}\cdot(\sigma'_\Psi,\, X_{N'+K-1},\, \cdots,\, X_{N'+K-1+|\Psi'|}$
Since $\lceil K\rceil^{M+|\Psi'|}_{N,K}$ is well-defined, we have that it does not contain variables larger than X_{N+K-1}, and thus we have:
$\lceil K\rceil^{M+|\Psi'|}_{N,K}\cdot(\sigma'_\Psi,\, X_{N'+K-1},\, \cdots,\, X_{N'+K-1+|\Psi'|} = \lceil K\rceil^{M+|\Psi'|}_{N,K}\cdot\sigma'_\Psi$.
Thus the two sides are equal.

Rest By direct application of the other parts.

Lemma B.104 (Binding of extension variables and extension substitution) *Assuming* $|\sigma_\Psi| = N$, $\lfloor\cdot\rfloor^M_{N,K}$ *and* $\lfloor\cdot\rfloor^M_{N',K}$ *are well-defined, we have:*

1. $\lfloor \mathbf{I} \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor \mathbf{I}\rfloor^M_{N,K} \cdot \sigma_\Psi$
2. $\lfloor t \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor t\rfloor^M_{N,K} \cdot \sigma_\Psi$
3. $\lfloor \Phi \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor \Phi\rfloor^M_{N,K} \cdot \sigma_\Psi$
4. $\lfloor \sigma \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor \sigma\rfloor^M_{N,K} \cdot \sigma_\Psi$
5. $\lfloor T \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor T\rfloor^M_{N,K} \cdot \sigma_\Psi$
6. $\lfloor K \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor K\rfloor^M_{N,K} \cdot \sigma_\Psi$
7. $\lfloor \Psi \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor \Psi\rfloor^M_{N,K} \cdot \sigma_\Psi$

Part 2 The interesting case is when $t = X_{N+j}/\sigma$ with $j < K$. In that case, the left-hand-side becomes: $\left\lfloor X_{N'+j}/(\sigma \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1}))\right\rfloor^M_{N',K} = B_{M+K-j-1}/(\lfloor \sigma \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K}) = B_M/(\lfloor\sigma\rfloor^M_{N,K} \cdot \sigma_\Psi)$ by part 4.
The right-hand-side becomes $(B_{M+K-j-1}/(\lfloor\sigma\rfloor^M_{N,K})) \cdot \sigma_\Psi = B_M/(\lfloor\sigma\rfloor^M_{N,K} \cdot \sigma_\Psi)$.

Rest Again, simple by induction and use of other parts; similarly as above.

Lemma B.105 1. $\left\lceil \lfloor \mathbf{I}\rfloor^M_{N,K}\right\rceil^M_{N,K} = \mathbf{I}$

2. $\left\lceil \lfloor t\rfloor^M_{N,K}\right\rceil^M_{N,K} = t$
3. $\left\lceil \lfloor \Phi\rfloor^M_{N,K}\right\rceil^M_{N,K} = \Phi$
4. $\left\lceil \lfloor \sigma\rfloor^M_{N,K}\right\rceil^M_{N,K} = \sigma$
5. $\left\lceil \lfloor T\rfloor^M_{N,K}\right\rceil^M_{N,K} = T$
6. $\left\lceil \lfloor K\rfloor^M_{N,K}\right\rceil^M_{N,K} = K$
7. $\left\lceil \lfloor \Psi\rfloor^M_{N,K}\right\rceil^M_{N,K} = \Psi$

Trivial by structural induction.

Lemma B.106 *If* $|\sigma_\Psi| = |\Psi|$ *and* $\uparrow\cdot\restriction_{|\Psi|}$ *and* $\uparrow\cdot\restriction_{|\Psi'|}$ *are well-defined, then* $\uparrow\Psi''\restriction_{|\Psi|} \cdot \sigma_\Psi = \uparrow\Psi'' \cdot \sigma_\Psi\restriction_{|\Psi'|}$

By induction on Ψ''.
When $\Psi'' = \bullet$, trivial.
When $\Psi'' = \Psi''', K$ we have that:
$\uparrow\Psi''', K\restriction_{|\Psi|} = \uparrow\Psi'''\restriction_{|\Psi|}, \lceil K\rceil_{|\Psi|,|\Psi'''|}$.
Applying σ_Ψ to this we get:
$\uparrow\Psi'''\restriction_{|\Psi|} \cdot \sigma_\Psi, \lceil K\rceil_{|\Psi|,|\Psi'''|} \cdot (\sigma_\Psi, X_{|\Psi'|}, \cdots, X_{|\Psi'|+|\Psi''|})$.
By induction hypothesis the first part is equal to:

$\lceil \Psi'' \cdot \sigma_\Psi \rceil_{|\Psi'|}$.
Using lemma B.87 for the second part we get that it's equal to:
$\lceil K \cdot \sigma_\Psi \rceil_{|\Psi'|,|\Psi''|}$
Furthermore, since K does not contain variables greater than $X_{|\Psi|}$, we have that $K \cdot \sigma_\Psi = K \cdot (\sigma_\Psi, X_{|\Psi'|}, \cdots, X_{|\Psi'|+|\Psi''|}$.
Thus, the left hand side is equal to $\left\lceil \Psi'' \cdot \sigma_\Psi, K \cdot (\sigma_\Psi, X_{|\Psi'|}, \cdots, X_{|\Psi'|+|\Psi''|} \right\rceil_{|\Psi'|,|\Psi''|}$, which is equal to the right-hand-side.

C. Definition and metatheory of computational language

Definition C.1 *The syntax of the computational language is defined below.*

$$
\begin{array}{rl}
k ::= & \star \mid k \to k \mid \Pi(K).k \\
\tau ::= & \Pi(K).\tau \mid \Sigma(K).\tau \mid \lambda(K).\tau \mid \tau\ T \\
& \mid \mathsf{unit} \mid \bot \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha : k.\tau \mid \mathsf{ref}\ \tau \mid \forall\alpha : k.\tau \mid \lambda\alpha : k.\tau \mid \tau_1\ \tau_2 \mid \alpha \\
e ::= & \Lambda(K).e \mid e\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ e \mid \mathsf{unpack}\ e\ (.)x.(e') \\
& \mid () \mid \mathsf{error} \mid \lambda x : \tau.e \mid e\ e' \mid x \mid (e, e') \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e \mid \mathsf{case}(e, x.e', x.e'') \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \mathsf{ref}\ e \\
& \mid e := e' \mid !e \mid l \mid \Lambda\alpha : k.e \mid e\ \tau \mid \mathsf{fix}\ x : \tau.e \\
& \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e') \\
\Gamma ::= & \bullet \mid \Gamma, x : \tau \mid \Gamma, \alpha : k \\
\Sigma ::= & \bullet \mid \Sigma, l : \tau
\end{array}
$$

Definition C.2 *Freshening and binding for computational kinds, types and terms are defined as follows.*

$\boxed{\lceil k \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil \star \rceil^M_{N,K} & = & \star \\
\lceil \Pi(K).k \rceil^M_{N,K} & = & \Pi(\lceil K \rceil^M_{N,K}).\ \lceil k \rceil^{M+1}_{N,K}
\end{array}
$$

$\boxed{\lceil \tau \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil \Pi(K).\tau \rceil^M_{N,K} & = & \Pi(\lceil K \rceil^M_{N,K}).\ \lceil \tau \rceil^{M+1}_{N,K} \\
\lceil \Sigma(K).\tau \rceil^M_{N,K} & = & \Sigma(\lceil K \rceil^M_{N,K}).\ \lceil \tau \rceil^{M+1}_{N,K} \\
\lceil \lambda(K).\tau \rceil^M_{N,K} & = & \lambda(\lceil K \rceil^M_{N,K}).\ \lceil \tau \rceil^{M+1}_{N,K} \\
\lceil \tau\ T \rceil^M_{N,K} & = & \lceil \tau \rceil^M_{N,K}\ \lceil T \rceil^M_{N,K} \\
\lceil \mathsf{unit} \rceil^M_{N,K} & = & \mathsf{unit} \\
\lceil \bot \rceil^M_{N,K} & = & \bot \\
\lceil \tau_1 \to \tau_2 \rceil^M_{N,K} & = & \lceil \tau_1 \rceil^M_{N,K} \to \lceil \tau_2 \rceil^M_{N,K} \\
\lceil \tau_1 \times \tau_2 \rceil^M_{N,K} & = & \lceil \tau_1 \rceil^M_{N,K} \times \lceil \tau_2 \rceil^M_{N,K} \\
\lceil \tau_1 + \tau_2 \rceil^M_{N,K} & = & \lceil \tau_1 \rceil^M_{N,K} + \lceil \tau_2 \rceil^M_{N,K} \\
\lceil \mu\alpha : k.\tau \rceil^M_{N,K} & = & \mu\alpha : \lceil k \rceil^M_{N,K}.\ \lceil \tau \rceil^M_{N,K} \\
\lceil \mathsf{ref}\ \tau \rceil^M_{N,K} & = & \mathsf{ref}\ \lceil \tau \rceil^M_{N,K} \\
\lceil \forall\alpha : k.\tau \rceil^M_{N,K} & = & \forall\alpha : \lceil k \rceil^M_{N,K}.\ \lceil \tau \rceil^M_{N,K} \\
\lceil \lambda\alpha : k.\tau \rceil^M_{N,K} & = & \lambda\alpha : \lceil k \rceil^M_{N,K}.\ \lceil \tau \rceil^M_{N,K} \\
\lceil \tau_1\ \tau_2 \rceil^M_{N,K} & = & \lceil \tau_1 \rceil^M_{N,K}\ \lceil \tau_2 \rceil^M_{N,K} \\
\lceil \alpha \rceil^M_{N,K} & = & \alpha
\end{array}
$$

$\boxed{\lceil e \rceil^M_{N,K}}$

$$
\begin{array}{lcl}
\lceil \Lambda(K).e \rceil^M_{N,K} & = & \Lambda(\lceil K \rceil^M_{N,K}).\, \lceil e \rceil^{M+1}_{N,K} \\
\lceil e\ T \rceil^M_{N,K} & = & \lceil e \rceil^M_{N,K}\ \lceil T \rceil^M_{N,K} \\
\lceil \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ e \rceil^M_{N,K} & = & \mathsf{pack}\ \lceil T \rceil^M_{N,K}\ \mathsf{return}\ (.\lceil \tau \rceil^{M+1}_{N,K})\ \mathsf{with}\ \lceil e \rceil^M_{N,K} \\
\lceil \mathsf{unpack}\ e\ (.)x.(e') \rceil^M_{N,K} & = & \mathsf{unpack}\ \lceil e \rceil^M_{N,K}\ (.)x.(\lceil e' \rceil^{M+1}_{N,K}) \\
\lceil () \rceil^M_{N,K} & = & () \\
\lceil \mathsf{error} \rceil^M_{N,K} & = & \mathsf{error} \\
\lceil \lambda x : \tau.e \rceil^M_{N,K} & = & \lambda x : \lceil \tau \rceil^M_{N,K}.\, \lceil e \rceil^M_{N,K} \\
\lceil e_1\ e_2 \rceil^M_{N,K} & = & \lceil e_1 \rceil^M_{N,K}\ \lceil e_2 \rceil^M_{N,K} \\
\lceil x \rceil^M_{N,K} & = & x \\
\lceil (e,\ e') \rceil^M_{N,K} & = & (\lceil e \rceil^M_{N,K},\ \lceil e' \rceil^M_{N,K}) \\
\lceil \mathsf{proj}_i\ e \rceil^M_{N,K} & = & \mathsf{proj}_i\ \lceil e \rceil^M_{N,K} \\
\lceil \mathsf{inj}_i\ e \rceil^M_{N,K} & = & \mathsf{inj}_i\ \lceil e \rceil^M_{N,K} \\
\lceil \mathsf{case}(e, x.e', x.e'') \rceil^M_{N,K} & = & \mathsf{case}(\lceil e \rceil^M_{N,K}, x.\lceil e' \rceil^M_{N,K}, x.\lceil e'' \rceil^M_{N,K}) \\
\lceil \mathsf{fold}\ e \rceil^M_{N,K} & = & \mathsf{fold}\ \lceil e \rceil^M_{N,K} \\
\lceil \mathsf{unfold}\ e \rceil^M_{N,K} & = & \mathsf{unfold}\ \lceil e \rceil^M_{N,K} \\
\lceil \mathsf{ref}\ e \rceil^M_{N,K} & = & \mathsf{ref}\ \lceil e \rceil^M_{N,K} \\
\lceil e_1 := e_2 \rceil^M_{N,K} & = & \lceil e_1 \rceil^M_{N,K} := \lceil e_2 \rceil^M_{N,K} \\
\lceil !e \rceil^M_{N,K} & = & !\lceil e \rceil^M_{N,K} \\
\lceil l \rceil^M_{N,K} & = & l \\
\lceil \Lambda\alpha : k.e \rceil^M_{N,K} & = & \Lambda\alpha : \lceil k \rceil^M_{N,K}.\, \lceil e \rceil^M_{N,K} \\
\lceil e\ \tau \rceil^M_{N,K} & = & \lceil e \rceil^M_{N,K}\ \lceil \tau \rceil^M_{N,K} \\
\lceil \mathsf{fix}\ x : \tau.e \rceil^M_{N,K} & = & \mathsf{fix}\ x : \lceil \tau \rceil^M_{N,K}.\, \lceil e \rceil^M_{N,K} \\
\lceil \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e') \rceil & = & \mathsf{unify}\ \lceil T \rceil^M_{N,K}\ \mathsf{return}\ (.\lceil \tau \rceil^{M+1}_{N,K})\ \mathsf{with}\ (\lceil \Psi \rceil^M_{N,K}.\lceil T' \rceil^{M+|\Psi|}_{N,K} \mapsto \lceil e' \rceil^{M+|\Psi|}_{N,K})
\end{array}
$$

$\boxed{\lfloor k \rfloor^M_{N,K}}$

$$
\begin{array}{lcl}
\lfloor \star \rfloor^M_{N,K} & = & \star \\
\lfloor \Pi(K).k \rfloor^M_{N,K} & = & \Pi(\lfloor K \rfloor^M_{N,K}).\, \lfloor k \rfloor^{M+1}_{N,K}
\end{array}
$$

$\boxed{\lfloor \tau \rfloor^M_{N,K}}$

$$
\begin{array}{lcl}
\lfloor \Pi(K).\tau \rfloor^M_{N,K} & = & \Pi(\lfloor K \rfloor^M_{N,K}).\, \lfloor \tau \rfloor^{M+1}_{N,K} \\
\lfloor \Sigma(K).\tau \rfloor^M_{N,K} & = & \Sigma(\lfloor K \rfloor^M_{N,K}).\, \lfloor \tau \rfloor^{M+1}_{N,K} \\
\lfloor \lambda(K).\tau \rfloor^M_{N,K} & = & \lambda(\lfloor K \rfloor^M_{N,K}).\, \lfloor \tau \rfloor^{M+1}_{N,K} \\
\lfloor \tau\ T \rfloor^M_{N,K} & = & \lfloor \tau \rfloor^M_{N,K}\ \lfloor T \rfloor^M_{N,K} \\
\lfloor \mathsf{unit} \rfloor^M_{N,K} & = & \mathsf{unit}
\end{array}
$$

$\boxed{\lfloor\tau\rfloor^M_{N,K} \text{ (continued)}}$

$$
\begin{array}{lcl}
\lfloor\bot\rfloor^M_{N,K} & = & \bot \\
\lfloor\tau_1 \to \tau_2\rfloor^M_{N,K} & = & \lfloor\tau_1\rfloor^M_{N,K} \to \lfloor\tau_2\rfloor^M_{N,K} \\
\lfloor\tau_1 \times \tau_2\rfloor^M_{N,K} & = & \lfloor\tau_1\rfloor^M_{N,K} \times \lfloor\tau_2\rfloor^M_{N,K} \\
\lfloor\tau_1 + \tau_2\rfloor^M_{N,K} & = & \lfloor\tau_1\rfloor^M_{N,K} + \lfloor\tau_2\rfloor^M_{N,K} \\
\lfloor\mu\alpha : k.\tau\rfloor^M_{N,K} & = & \mu\alpha : \lfloor k\rfloor^M_{N,K} . \lfloor\tau\rfloor^M_{N,K} \\
\lfloor\mathsf{ref}\ \tau\rfloor^M_{N,K} & = & \mathsf{ref}\ \lfloor\tau\rfloor^M_{N,K} \\
\lfloor\forall\alpha : k.\tau\rfloor^M_{N,K} & = & \forall\alpha : \lfloor k\rfloor^M_{N,K} . \lfloor\tau\rfloor^M_{N,K} \\
\lfloor\lambda\alpha : k.\tau\rfloor^M_{N,K} & = & \lambda\alpha : \lfloor k\rfloor^M_{N,K} . \lfloor\tau\rfloor^M_{N,K} \\
\lfloor\tau_1\ \tau_2\rfloor^M_{N,K} & = & \lfloor\tau_1\rfloor^M_{N,K}\ \lfloor\tau_2\rfloor^M_{N,K} \\
\lfloor\alpha\rfloor^M_{N,K} & = & \alpha
\end{array}
$$

$\boxed{\lfloor e\rfloor^M_{N,K}}$

$$
\begin{array}{lcl}
\lfloor\Lambda(K).e\rfloor^M_{N,K} & = & \Lambda(\lfloor K\rfloor^M_{N,K}) . \lfloor e\rfloor^{M+1}_{N,K} \\
\lfloor e\ T\rfloor^M_{N,K} & = & \lfloor e\rfloor^M_{N,K}\ \lfloor T\rfloor^M_{N,K} \\
\lfloor\mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ e\rfloor^M_{N,K} & = & \mathsf{pack}\ \lfloor T\rfloor^M_{N,K}\ \mathsf{return}\ (.\lfloor\tau\rfloor^{M+1}_{N,K})\ \mathsf{with}\ \lfloor e\rfloor^M_{N,K} \\
\lfloor\mathsf{unpack}\ e\ (.)x.(e')\rfloor^M_{N,K} & = & \mathsf{unpack}\ \lfloor e\rfloor^M_{N,K}\ (.)x.(\lfloor e'\rfloor^{M+1}_{N,K}) \\
\lfloor()\rfloor^M_{N,K} & = & () \\
\lfloor\mathsf{error}\rfloor^M_{N,K} & = & \mathsf{error} \\
\lfloor\lambda x : \tau.e\rfloor^M_{N,K} & = & \lambda x : \lfloor\tau\rfloor^M_{N,K} . \lfloor e\rfloor^M_{N,K} \\
\lfloor e_1\ e_2\rfloor^M_{N,K} & = & \lfloor e_1\rfloor^M_{N,K}\ \lfloor e_2\rfloor^M_{N,K} \\
\lfloor x\rfloor^M_{N,K} & = & x \\
\lfloor(e,\ e')\rfloor^M_{N,K} & = & (\lfloor e\rfloor^M_{N,K},\ \lfloor e'\rfloor^M_{N,K}) \\
\lfloor\mathsf{proj}_i\ e\rfloor^M_{N,K} & = & \mathsf{proj}_i\ \lfloor e\rfloor^M_{N,K} \\
\lfloor\mathsf{inj}_i\ e\rfloor^M_{N,K} & = & \mathsf{inj}_i\ \lfloor e\rfloor^M_{N,K} \\
\lfloor\mathsf{case}(e, x.e', x.e'')\rfloor^M_{N,K} & = & \mathsf{case}(\lfloor e\rfloor^M_{N,K}, x.\lfloor e'\rfloor^M_{N,K}, x.\lfloor e''\rfloor^M_{N,K}) \\
\lfloor\mathsf{fold}\ e\rfloor^M_{N,K} & = & \mathsf{fold}\ \lfloor e\rfloor^M_{N,K} \\
\lfloor\mathsf{unfold}\ e\rfloor^M_{N,K} & = & \mathsf{unfold}\ \lfloor e\rfloor^M_{N,K} \\
\lfloor\mathsf{ref}\ e\rfloor^M_{N,K} & = & \mathsf{ref}\ \lfloor e\rfloor^M_{N,K} \\
\lfloor e_1 := e_2\rfloor^M_{N,K} & = & \lfloor e_1\rfloor^M_{N,K} := \lfloor e_2\rfloor^M_{N,K} \\
\lfloor !e\rfloor^M_{N,K} & = & !\lfloor e\rfloor^M_{N,K} \\
\lfloor l\rfloor^M_{N,K} & = & l \\
\lfloor\Lambda\alpha : k.e\rfloor^M_{N,K} & = & \Lambda\alpha : \lfloor k\rfloor^M_{N,K} . \lfloor e\rfloor^M_{N,K} \\
\lfloor e\ \tau\rfloor^M_{N,K} & = & \lfloor e\rfloor^M_{N,K}\ \lfloor\tau\rfloor^M_{N,K} \\
\lfloor\mathsf{fix}\ x : \tau.e\rfloor^M_{N,K} & = & \mathsf{fix}\ x : \lfloor\tau\rfloor^M_{N,K} . \lfloor e\rfloor^M_{N,K} \\
\lfloor\mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e')\rfloor & = & \mathsf{unify}\ \lfloor T\rfloor^M_{N,K}\ \mathsf{return}\ (.\lfloor\tau\rfloor^{M+1}_{N,K})\ \mathsf{with}\ (\lfloor\Psi\rfloor^M_{N,K} . \lfloor T'\rfloor^{M+|\Psi|}_{N,K} \mapsto \lfloor e'\rfloor^{M+|\Psi|}_{N,K})
\end{array}
$$

Definition C.3 *Extension substitution application to computational-level kinds, types and terms.*

$\boxed{k \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
\star \cdot \sigma_\Psi & = & \star \\
(k \to k) \cdot \sigma_\Psi & = & k \cdot \sigma_\Psi \to k \cdot \sigma_\Psi \\
(\Pi(K).k) \cdot \sigma_\Psi & = & \Pi(K \cdot \sigma_\Psi).k \cdot \sigma_\Psi
\end{array}
$$

$\boxed{\tau \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
(\Pi(K).\tau) \cdot \sigma_\Psi & = & \Pi(K \cdot \sigma_\Psi).\tau \cdot \sigma_\Psi \\
(\Sigma(K).\tau) \cdot \sigma_\Psi & = & \Sigma(K \cdot \sigma_\Psi).\tau \cdot \sigma_\Psi \\
(\lambda(K).\tau) \cdot \sigma_\Psi & = & \lambda(K \cdot \sigma_\Psi).\tau \cdot \sigma_\Psi \\
(\tau\ T) \cdot \sigma_\Psi & = & \tau \cdot \sigma_\Psi\ T \cdot \sigma_\Psi \\
\mathsf{unit} \cdot \sigma_\Psi & = & \mathsf{unit} \\
\bot \cdot \sigma_\Psi & = & \bot \\
(\tau_1 \to \tau_2) \cdot \sigma_\Psi & = & \tau_1 \cdot \sigma_\Psi \to \tau_2 \cdot \sigma_\Psi \\
(\tau_1 \times \tau_2) \cdot \sigma_\Psi & = & \tau_1 \cdot \sigma_\Psi \times \tau_2 \cdot \sigma_\Psi \\
(\tau_1 + \tau_2) \cdot \sigma_\Psi & = & \tau_1 \cdot \sigma_\Psi + \tau_2 \cdot \sigma_\Psi \\
(\mu\alpha : k.\tau) \cdot \sigma_\Psi & = & \mu\alpha : k \cdot \sigma_\Psi.\tau \cdot \sigma_\Psi \\
(\mathsf{ref}\ \tau) \cdot \sigma_\Psi & = & \mathsf{ref}\ \tau \cdot \sigma_\Psi \\
(\forall\alpha : k.\tau) \cdot \sigma_\Psi & = & \forall\alpha : k \cdot \sigma_\Psi.\tau \cdot \sigma_\Psi \\
(\lambda\alpha : k.\tau) \cdot \sigma_\Psi & = & \lambda\alpha : k \cdot \sigma_\Psi.\tau \cdot \sigma_\Psi \\
(\tau_1\ \tau_2) \cdot \sigma_\Psi & = & \tau_1 \cdot \sigma_\Psi\ \tau_2 \cdot \sigma_\Psi \\
\alpha \cdot \sigma_\Psi & = & \alpha
\end{array}
$$

$\boxed{e \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
(\Lambda(K).e) \cdot \sigma_\Psi & = & \Lambda(K \cdot \sigma_\Psi).e \cdot \sigma_\Psi \\
(e\ T) \cdot \sigma_\Psi & = & e \cdot \sigma_\Psi\ T \cdot \sigma_\Psi \\
(\mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ e) \cdot \sigma_\Psi & = & \mathsf{pack}\ T \cdot \sigma_\Psi\ \mathsf{return}\ (.\tau \cdot \sigma_\Psi)\ \mathsf{with}\ e \cdot \sigma_\Psi \\
(\mathsf{unpack}\ e\ (.)x.(e')) \cdot \sigma_\Psi & = & \mathsf{unpack}\ e \cdot \sigma_\Psi\ (.)x.(e' \cdot \sigma_\Psi) \\
() \cdot \sigma_\Psi & = & () \\
\mathsf{error} \cdot \sigma_\Psi & = & \mathsf{error} \\
(\lambda x : \tau.e) \cdot \sigma_\Psi & = & \lambda x : \tau \cdot \sigma_\Psi.e \cdot \sigma_\Psi \\
(e\ e') \cdot \sigma_\Psi & = & e \cdot \sigma_\Psi\ e' \cdot \sigma_\Psi \\
x \cdot \sigma_\Psi & = & x \\
(e,\ e') \cdot \sigma_\Psi & = & (e \cdot \sigma_\Psi,\ e' \cdot \sigma_\Psi) \\
(\mathsf{proj}_i\ e) \cdot \sigma_\Psi & = & \mathsf{proj}_i\ e \cdot \sigma_\Psi \\
(\mathsf{inj}_i\ e) \cdot \sigma_\Psi & = & \mathsf{inj}_i\ e \cdot \sigma_\Psi \\
(\mathsf{case}(e, x.e', x.e'')) \cdot \sigma_\Psi & = & \mathsf{case}(e \cdot \sigma_\Psi, x.e' \cdot \sigma_\Psi, x.e'' \cdot \sigma_\Psi) \\
(\mathsf{fold}\ e) \cdot \sigma_\Psi & = & \mathsf{fold}\ e \cdot \sigma_\Psi \\
(\mathsf{unfold}\ e) \cdot \sigma_\Psi & = & \mathsf{unfold}\ e \cdot \sigma_\Psi \\
(\mathsf{ref}\ e) \cdot \sigma_\Psi & = & \mathsf{ref}\ e \cdot \sigma_\Psi \\
(e := e') \cdot \sigma_\Psi & = & e \cdot \sigma_\Psi := e' \cdot \sigma_\Psi \\
(!e) \cdot \sigma_\Psi & = & !e \cdot \sigma_\Psi \\
l \cdot \sigma_\Psi & = & l \\
(\Lambda\alpha : k.e) \cdot \sigma_\Psi & = & \Lambda\alpha : k \cdot \sigma_\Psi.e \cdot \sigma_\Psi \\
(e\ \tau) \cdot \sigma_\Psi & = & e \cdot \sigma_\Psi\ \tau \cdot \sigma_\Psi \\
(\mathsf{fix}\ x : \tau.e) \cdot \sigma_\Psi & = & \mathsf{fix}\ x : \tau \cdot \sigma_\Psi.e \cdot \sigma_\Psi \\
(\mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e')) \cdot \sigma_\Psi & = & \mathsf{unify}\ T \cdot \sigma_\Psi\ \mathsf{return}\ (.\tau \cdot \sigma_\Psi)\ \mathsf{with}\ (\Psi \cdot \sigma_\Psi.T' \cdot \sigma_\Psi \mapsto e' \cdot \sigma_\Psi)
\end{array}
$$

$\boxed{\Gamma \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\Gamma,\ x : \tau) \cdot \sigma_\Psi & = & \Gamma \cdot \sigma_\Psi,\ x : \tau \cdot \sigma_\Psi \\
(\Gamma,\ \alpha : k) \cdot \sigma_\Psi & = & \Gamma \cdot \sigma_\Psi,\ \alpha : k \cdot \sigma_\Psi
\end{array}
$$

Definition C.4 *The typing judgements for the computational language are given below.*

$\boxed{\Psi \vdash k \text{ wf}}$

$$\frac{\vdash \Psi \text{ wf}}{\Psi \vdash \star \text{ wf}} \qquad \frac{\Psi \vdash k \text{ wf} \qquad \Psi \vdash k' \text{ wf}}{\Psi \vdash k \to k' \text{ wf}} \qquad \frac{\vdash \Psi,\, K \text{ wf} \qquad \Psi,\, K \vdash \lceil k \rceil_{|\Psi|,1} \text{ wf}}{\Psi \vdash \Pi(K).k \text{ wf}}$$

$\boxed{\Psi;\, \Gamma \vdash \tau : k}$

$$\frac{\Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star}{\Psi;\, \Gamma \vdash \Pi(K).\tau : \star} \qquad \frac{\Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star}{\Psi;\, \Gamma \vdash \Sigma(K).\tau : \star} \qquad \frac{\Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : k}{\Psi;\, \Gamma \vdash \lambda(K).\tau : \Pi(K).\, \lfloor k \rfloor_{|\Psi|,1}}$$

$$\frac{\Psi;\, \Gamma \vdash \tau : \Pi(K).k \qquad \Psi \vdash T : K}{\Psi;\, \Gamma \vdash \tau\, T : \lceil k \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)} \qquad \frac{}{\Psi;\, \Gamma \vdash \mathsf{unit} : \star} \qquad \frac{}{\Psi;\, \Gamma \vdash \bot : \star} \qquad \frac{\Psi;\, \Gamma \vdash \tau_1 : \star \qquad \Psi;\, \Gamma \vdash \tau_2 : \star}{\Psi;\, \Gamma \vdash \tau_1 \to \tau_2 : \star}$$

$$\frac{\Psi;\, \Gamma \vdash \tau_1 : \star \qquad \Psi;\, \Gamma \vdash \tau_2 : \star}{\Psi;\, \Gamma \vdash \tau_1 \times \tau_2 : \star} \qquad \frac{\Psi;\, \Gamma \vdash \tau_1 : \star \qquad \Psi;\, \Gamma \vdash \tau_2 : \star}{\Psi;\, \Gamma \vdash \tau_1 + \tau_2 : \star} \qquad \frac{\Psi \vdash k \text{ wf} \qquad \Psi;\, \Gamma,\, \alpha : k \vdash \tau : k}{\Psi;\, \Gamma \vdash \mu\alpha : k.\tau : k}$$

$$\frac{\Psi;\, \Gamma \vdash \tau : \star}{\Psi;\, \Gamma \vdash \mathsf{ref}\, \tau : \star} \qquad \frac{\Psi \vdash k \text{ wf} \qquad \Psi;\, \Gamma,\, \alpha : k \vdash \tau : \star}{\Psi;\, \Gamma \vdash \forall\alpha : k.\tau : \star} \qquad \frac{\Psi \vdash k \text{ wf} \qquad \Psi;\, \Gamma,\, \alpha : k \vdash \tau : k'}{\Psi;\, \Gamma \vdash \lambda\alpha : k.\tau : k \to k'}$$

$$\frac{\Psi;\, \Gamma \vdash \tau_1 : k \to k' \qquad \Psi;\, \Gamma \vdash \tau_2 : k}{\Psi;\, \Gamma \vdash \tau_1\, \tau_2 : k'} \qquad \frac{(\alpha : k) \in \Gamma}{\Psi;\, \Gamma \vdash \alpha : k}$$

$\boxed{\Psi;\, \Sigma;\, \Gamma \vdash e : \tau}$

$$\frac{\Psi,\, K;\, \Sigma;\, \Gamma \vdash \lceil e \rceil_{|\Psi|,1} : \tau}{\Psi;\, \Sigma;\, \Gamma \vdash \Lambda(K).e : \Pi(K).\, \lfloor \tau \rfloor_{|\Psi|,1}} \qquad \frac{\Psi;\, \Sigma;\, \Gamma \vdash e : \Pi(K).\tau \qquad \Psi \vdash T : K}{\Psi;\, \Sigma;\, \Gamma \vdash e\, T : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)}$$

$$\frac{\Psi \vdash T : K \qquad \Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star \qquad \Psi;\, \Sigma;\, \Gamma \vdash e : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{pack}\, T\, \mathsf{return}\, (.\tau)\, \mathsf{with}\, e : \Sigma(K).\tau}$$

$$\frac{\Psi;\, \Sigma;\, \Gamma \vdash e : \Sigma(K).\tau \qquad \Psi,\, K,\, \Sigma;\, \Gamma,\, x : \lceil \tau \rceil_{|\Psi|,1} \vdash \lceil e' \rceil_{|\Psi|,1} : \tau' \qquad \Psi;\, \Gamma \vdash \tau' : \star}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{unpack}\, e\, (.)x.(e') : \tau'} \qquad \frac{}{\Psi;\, \Sigma;\, \Gamma \vdash () : \mathsf{unit}}$$

$$\frac{}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{error} : \tau} \qquad \frac{\Psi;\, \Sigma;\, \Gamma,\, x : \tau \vdash e : \tau'}{\Psi;\, \Sigma;\, \Gamma \vdash \lambda x : \tau.e : \tau \to \tau'}$$

$$\frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \tau \to \tau' \qquad \Psi;\,\Sigma;\,\Gamma \vdash e' : \tau}{\Psi;\,\Sigma;\,\Gamma \vdash e\ e' : \tau'} \qquad \frac{(x : \tau) \in \Gamma}{\Psi;\,\Sigma;\,\Gamma \vdash x : \tau} \qquad \frac{\Psi;\,\Sigma;\,\Gamma \vdash e_1 : \tau_1 \qquad \Psi;\,\Sigma;\,\Gamma \vdash e_2 : \tau_2}{\Psi;\,\Sigma;\,\Gamma \vdash (e_1,\, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \tau_1 \times \tau_2 \qquad i = 1 \text{ or } 2}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{proj}_i\ e : \tau_i} \qquad \frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \tau_i \qquad i = 1 \text{ or } 2}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{inj}_i\ e : \tau_1 + \tau_2}$$

$$\frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Psi;\,\Sigma;\,\Gamma,\, x : \tau_1 \vdash e_1 : \tau \qquad \Psi;\,\Sigma;\,\Gamma,\, x : \tau_2 \vdash e_2 : \tau}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{case}(e, x.e_1, x.e_2) : \tau}$$

$$\frac{\Psi;\,\Gamma \vdash \mu\alpha : k.\tau : k \qquad \Psi;\,\Sigma;\,\Gamma \vdash e : \tau[\mu\alpha : k.\tau/\alpha]\ \tau_1\ \tau_2\ \cdots\ \tau_n}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{fold}\ e : (\mu\alpha : k.\tau)\ \tau_1\ \tau_2\ \cdots\ \tau_n}$$

$$\frac{\Psi;\,\Gamma \vdash \mu\alpha : k.\tau : k \qquad \Psi;\,\Sigma;\,\Gamma \vdash e : (\mu\alpha : k.\tau)\ \tau_1\ \tau_2\ \cdots\ \tau_n}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{unfold}\ e : \tau[\mu\alpha : k.\tau/\alpha]\ \tau_1\ \tau_2\ \cdots\ \tau_n} \qquad \frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \tau}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau}$$

$$\frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \mathsf{ref}\ \tau \qquad \Psi;\,\Sigma;\,\Gamma \vdash e' : \tau}{\Psi;\,\Sigma;\,\Gamma \vdash e := e' : \mathsf{unit}} \qquad \frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \mathsf{ref}\ \tau}{\Psi;\,\Sigma;\,\Gamma \vdash !e : \tau} \qquad \frac{(l : \tau) \in \Sigma}{\Psi;\,\Sigma;\,\Gamma \vdash l : \mathsf{ref}\ \tau}$$

$$\frac{\Psi;\,\Sigma;\,\Gamma,\, \alpha : k \vdash e : \tau}{\Psi;\,\Sigma;\,\Gamma \vdash \Lambda\alpha : k.e : \Pi\alpha : k.\tau} \qquad \frac{\Psi;\,\Sigma;\,\Gamma \vdash e : \Pi\alpha : k.\tau' \qquad \Psi;\,\Gamma \vdash \tau : k}{\Psi;\,\Sigma;\,\Gamma \vdash e\ \tau : \tau'[\tau/\alpha]} \qquad \frac{\Psi;\,\Sigma;\,\Gamma,\, x : \tau \vdash e : \tau}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{fix}\ x : \tau.e : \tau}$$

$$\frac{\begin{array}{c}\Psi \vdash T : K \qquad \Psi,\, K;\,\Gamma \vdash \lceil\tau\rceil_{|\Psi|,1} : \star \qquad \Psi \vdash \lceil\Psi'\rceil_{|\Psi|}\ \mathrm{wf} \\ \Psi,\, \lceil\Psi'\rceil_{|\Psi|} \vdash \lceil T'\rceil_{|\Psi|,|\Psi'|} : K \qquad \Psi,\, \lceil\Psi'\rceil_{|\Psi|};\,\Sigma;\,\Gamma \vdash \lceil e'\rceil_{|\Psi|,|\Psi'|} : \lceil\tau\rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, \lceil T'\rceil_{|\Psi|,|\Psi'|})\end{array}}{\Psi;\,\Sigma;\,\Gamma \vdash \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi'.T' \mapsto e') : (\lceil\tau\rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)) + \mathsf{unit}}$$

$\boxed{\Psi \vdash \Gamma\ \mathrm{wf}}$

$$\frac{}{\Psi \vdash \bullet\ \mathrm{wf}} \qquad \frac{\Psi \vdash \Gamma\ \mathrm{wf} \qquad \Psi;\,\Gamma \vdash k\ \mathrm{wf}}{\Psi \vdash (\Gamma,\, \alpha : k)\ \mathrm{wf}} \qquad \frac{\Psi \vdash \Gamma\ \mathrm{wf} \qquad \Psi;\,\Gamma \vdash \tau : \star}{\Psi \vdash (\Gamma,\, x : \tau)\ \mathrm{wf}}$$

$\boxed{\vdash \Sigma\ \mathrm{wf}}$

$$\frac{}{\vdash \bullet\ \mathrm{wf}} \qquad \frac{\vdash \Sigma\ \mathrm{wf} \qquad \bullet;\,\bullet \vdash \tau : \star}{\vdash (\Sigma,\, l : \tau)}$$

Definition C.5 β*-equivalence for types* τ *is the symmetric, reflexive, transitive congruence closure of the following relation. Types of the language are viewed implicitly up to* β*-equivalence. This means that the lemmas that we prove about types need to agree on* β*-equivalent types.*

$$(\lambda\alpha : \mathcal{K}.\tau)\ \tau' \quad = \quad \tau[\tau'/\alpha]$$

Definition C.6 *Small-step operational semantics for the language are defined below.*

$$\begin{array}{l} v ::= \Lambda(K).e \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ v \mid () \mid \lambda x : \tau.e \mid (v,\, v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e \\ \mathcal{E} ::= \bullet \mid \mathcal{E}\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ \mathcal{E} \mid \mathsf{unpack}\ \mathcal{E}\ (.)x.(e') \mid \mathcal{E}\ e' \mid v\ \mathcal{E} \mid (\mathcal{E},\, e) \mid (v,\, \mathcal{E}) \mid \mathsf{proj}_i\ \mathcal{E} \mid \mathsf{inj}_i\ \mathcal{E} \\ \qquad \mid \mathsf{case}(\mathcal{E}, x.e_1, x.e_2) \mid \mathsf{fold}\ \mathcal{E} \mid \mathsf{unfold}\ \mathcal{E} \mid \mathsf{ref}\ \mathcal{E} \mid \mathcal{E} := e' \mid v := \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E}\ \tau \\ \mu ::= \bullet \mid \mu,\, l \mapsto v \end{array}$$

$\boxed{(\,\mu\,,\,e\,) \longrightarrow ((\,\mu\,,\,e'\,)|\mathsf{error})}$

$$\frac{(\,\mu\,,\,e\,) \longrightarrow (\,\mu'\,,\,e'\,)}{(\,\mu\,,\,\mathcal{E}[e]\,) \longrightarrow (\,\mu'\,,\,\mathcal{E}[e']\,)} \qquad (\,\mu\,,\,\mathcal{E}[\mathsf{error}]\,) \longrightarrow \mathsf{error} \qquad (\,\mu\,,\,(\Lambda(K).e)\;T\,) \longrightarrow (\,\mu\,,\,\lceil e\rceil_{0,1}\cdot T\,)$$

$$(\,\mu\,,\,\mathsf{unpack}\;\langle T,\,\tau\rangle\,v\,(.)x.(e')\,) \longrightarrow (\,\mu\,,\,(\lceil e'\rceil_{0,1}\cdot T)[v/x]\,) \qquad (\,\mu\,,\,(\lambda x:\tau.e)\;v\,) \longrightarrow (\,\mu\,,\,e[v/x]\,)$$

$$(\,\mu\,,\,\mathsf{proj}_i(v_1,\,v_2)\,) \longrightarrow (\,\mu\,,\,v_i\,) \qquad (\,\mu\,,\,\mathsf{case}(\mathsf{inj}_i\;v, x.e_1, x.e_2)\,) \longrightarrow (\,\mu\,,\,e_i[v/x]\,)$$

$$(\,\mu\,,\,\mathsf{unfold}\;(\mathsf{fold}\;v)\,) \longrightarrow (\,\mu\,,\,v\,) \qquad \frac{\neg(l \mapsto \_ \in \mu)}{(\,\mu\,,\,\mathsf{ref}\;v\,) \longrightarrow (\,(\mu,\,l\mapsto v)\,,\,l\,)} \qquad \frac{l \mapsto \_ \in \mu}{(\,\mu\,,\,l := v\,) \longrightarrow (\,\mu[l\mapsto v]\,,\,()\,)}$$

$$\frac{l \mapsto v \in \mu}{(\,\mu\,,\,!l\,) \longrightarrow (\,\mu\,,\,v\,)} \qquad (\,\mu\,,\,(\Lambda\alpha:k.e)\;\tau\,) \longrightarrow (\,\mu\,,\,e[\tau/\alpha]\,) \qquad (\,\mu\,,\,\mathsf{fix}\;x:\tau.e\,) \longrightarrow (\,\mu\,,\,e[\mathsf{fix}\;x:\tau.e/x]\,)$$

$$\frac{\exists\sigma_\Psi.(\bullet \vdash \sigma_\Psi : \lceil\Psi\rceil_0 \quad\wedge\quad \lceil T'\rceil_{0,|\Psi|}\cdot\sigma_\Psi = T)}{(\,\mu\,,\,\mathsf{unify}\;T\;\mathsf{return}\;(.\tau)\;\mathsf{with}\;(\Psi.T'\mapsto e')\,) \longrightarrow (\,\mu\,,\,\mathsf{inj}_1\;(\lceil e'\rceil_{0,|\Psi|}\cdot\sigma_\Psi)\,)}$$

$$\frac{\nexists\sigma_\Psi.(\bullet \vdash \sigma_\Psi : \lceil\Psi\rceil_0 \quad\wedge\quad \lceil T'\rceil_{0,|\Psi|}\cdot\sigma_\Psi = T)}{(\,\mu\,,\,\mathsf{unify}\;T\;\mathsf{return}\;(.\tau)\;\mathsf{with}\;(\Psi.T'\mapsto e')\,) \longrightarrow (\,\mu\,,\,\mathsf{inj}_2\;()\,)}$$

$\boxed{(l\mapsto v)\in\mu}$

$$\begin{array}{rcl}(l\mapsto v) & \in & (\mu,\,l\mapsto v)\\ (l\mapsto v) & \in & (\mu,\,l'\mapsto v') \Leftarrow (l\mapsto v)\in\mu\end{array}$$

$\boxed{(l:\tau)\in\Sigma}$

$$\begin{array}{rcl}(l:\tau) & \in & (\Sigma,\,l:\tau)\\ (l:\tau) & \in & (\Sigma,\,l':\tau') \Leftarrow (l:\tau)\in\Sigma\end{array}$$

$\boxed{\mu\sim\Sigma}$

$$\begin{array}{rcl}(l\mapsto v)\in\mu & \Rightarrow & \exists\tau.(l:\tau)\in\Sigma\wedge\bullet;\,\Sigma;\,\bullet\vdash v:\tau\\ (l:\tau)\in\Sigma & \Rightarrow & \exists v.(l\mapsto v)\in\mu\wedge\bullet;\,\Sigma;\,\bullet\vdash v:\tau\end{array}$$

$\boxed{\Sigma\subseteq\Sigma'}$

$$(l:\tau)\in\Sigma \quad\Rightarrow\quad (l:\tau)\in\Sigma'$$

$\boxed{\mu[l:=v]}$

$$\begin{array}{rcl}(\mu,\,l'\mapsto v')[l:=v] & = & \mu[l:=v],\,l'\mapsto v'\\ (\mu,\,l\mapsto v')[l:=v] & = & \mu, l\mapsto v\end{array}$$

Lemma C.7 (Computational substitution commutes with logic operations) *1.* $\lceil\tau[\tau'/\alpha]\rceil^M_{N,K} = \lceil\tau\rceil^M_{N,K}[\lceil\tau'\rceil^M_{N,K}/\alpha]$

2. $\lfloor \tau[\tau'/\alpha] \rfloor^M_{N,K} = \lfloor \tau \rfloor^M_{N,K}[\lfloor \tau' \rfloor^M_{N,K}/\alpha]$
3. $(\tau[\tau'/\alpha]) \cdot \sigma_\Psi = \tau \cdot \sigma_\Psi[\tau' \cdot \sigma_\Psi/\alpha]$
4. $\lceil e[\tau/\alpha] \rceil^M_{N,K} = \lceil e \rceil^M_{N,K}[\lceil \tau \rceil^M_{N,K}/\alpha]$
5. $\lfloor e[\tau/\alpha] \rfloor^M_{N,K} = \lfloor e \rfloor^M_{N,K}[\lfloor \tau \rfloor^M_{N,K}/\alpha]$
6. $(e[\tau/\alpha]) \cdot \sigma_\Psi = e \cdot \sigma_\Psi[\tau \cdot \sigma_\Psi/\alpha]$
7. $\lceil e[e'/x] \rceil^M_{N,K} = \lceil e \rceil^M_{N,K}[\lceil e' \rceil^M_{N,K}/x]$
8. $\lfloor e[e'/x] \rfloor^M_{N,K} = \lfloor e \rfloor^M_{N,K}[\lfloor e' \rfloor^M_{N,K}/x]$
9. $(e[e'/x]) \cdot \sigma_\Psi = e \cdot \sigma_\Psi[e' \cdot \sigma_\Psi/x]$

Simple by induction.

Lemma C.8 (Compatibility of β conversion with logic operations) *If $\tau =_\beta \tau'$ then:*

1. $\lceil \tau \rceil^M_{N,K} =_\beta \lceil \tau' \rceil^M_{N,K}$
2. $\lfloor \tau \rfloor^M_{N,K} =_\beta \lceil \tau' \rceil^M_{N,K}$
3. $\tau \cdot \sigma_\Psi =_\beta \tau' \cdot \sigma_\Psi$

In all cases it's trivially provable by expansion for $\tau = (\lambda\alpha : k.\tau_1)\tau_2$ and $\tau' = \tau_1[\tau_2/\alpha]$ and use of lemma C.7. The congruence cases for other τ, τ' are provable by induction hypothesis.

Lemma C.9 *Assuming $|\sigma_\Psi| = N$, $\lceil \cdot \rceil^M_{N,K}$ and $\lceil \cdot \rceil^M_{N',K}$ are well-defined for their respective arguments, we have:*

1. $\lceil k \cdot \sigma_\Psi \rceil^M_{N',K} = \lceil k \rceil^M_{N,K} \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$
2. $\lceil \tau \cdot \sigma_\Psi \rceil^M_{N',K} = \lceil \tau \rceil^M_{N,K} \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$
3. $\lceil e \cdot \sigma_\Psi \rceil^M_{N',K} = \lceil e \rceil^M_{N,K} \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$

By structural induction. We prove only the interesting cases.

Part 1 When $k = \Pi(K).k'$, we have that the left-hand-side is equal to:
$\lceil \Pi(K \cdot \sigma_\Psi).(k' \cdot \sigma_\Psi) \rceil^M_{N',K} = \Pi(\lceil K \cdot \sigma_\Psi \rceil^M_{N',K}).\lceil k' \cdot \sigma_\Psi \rceil^{M+1}_{N',K}$
We have by lemma B.103 that $\lceil K \cdot \sigma_\Psi \rceil^M_{N',K} = \lceil K \rceil^M_{N,K} \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$.
By induction hypothesis we have that $\lceil k' \cdot \sigma_\Psi \rceil^{M+1}_{N',K} = \lceil k' \rceil^{M+1}_{N,K} \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})$.
After expansion of freshening for the right-hand-side, we see that it is equal to the above.

Part 3 The most interesting case occurs when $e = \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e')$. We have that the left-hand-side is equal to:
$\lceil \mathsf{unify}\ T \cdot \sigma_\Psi\ \mathsf{return}\ (.\tau \cdot \sigma_\Psi)\ \mathsf{with}\ (\Psi \cdot \sigma_\Psi.T' \cdot \sigma_\Psi \mapsto e' \cdot \sigma_\Psi) \rceil^M_{N',K}$
By expansion of the definition of freshening we get that this is equal to:
$\mathsf{unify}\ \lceil T \cdot \sigma_\Psi \rceil^M_{N',K}\ \mathsf{return}\ (.\lceil \tau \cdot \sigma_\Psi \rceil^{M+1}_{N',K})\ \mathsf{with}\ (\lceil \Psi \cdot \sigma_\Psi \rceil^M_{N',K}.\lceil T' \rceil^{M+|\Psi \cdot \sigma_\Psi|}_{N,K} \mapsto \lceil e' \rceil^{M+|\Psi \cdot \sigma_\Psi|}_{N',K})$
The right-hand-side is equal to:
(assuming $\sigma'_\Psi = \sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1}$)
$\mathsf{unify}\ \lceil T \rceil^M_{N,K} \cdot \sigma'_\Psi\ \mathsf{return}\ (.\lceil \tau \rceil^{M+1}_{N,K} \cdot \sigma'_\Psi)\ \mathsf{with}\ (\lceil \Psi \rceil^M_{N,K} \cdot \sigma'_\Psi.\lceil T' \rceil^{M+|\Psi|}_{N,K} \cdot \sigma'_\Psi \mapsto \lceil e' \rceil^{M+|\Psi|}_{N,K} \cdot \sigma'_\Psi)$
In all cases, the respective terms match, by use of induction hypothesis, lemma B.103, and also the fact that $|\Psi| = |\Psi \cdot \sigma_\Psi|$.

Lemma C.10 *Assuming $|\sigma_\Psi| = N$, $\lfloor\cdot\rfloor^M_{N,K}$ and $\lfloor\cdot\rfloor^M_{N',K}$ are well-defined for their respective arguments, we have:*

1. $\lfloor k \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor k\rfloor^M_{N,K} \cdot \sigma_\Psi$
2. $\lfloor \tau \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor \tau\rfloor^M_{N,K} \cdot \sigma_\Psi$
3. $\lfloor e \cdot (\sigma_\Psi, X_{N'}, \cdots, X_{N'+K-1})\rfloor^M_{N',K} = \lfloor e\rfloor^M_{N,K} \cdot \sigma_\Psi$

Similarly as above, and use of lemma B.104.

Lemma C.11 1. $(k \cdot \sigma_\Psi) \cdot \sigma'_\Psi = k \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$
2. $(\tau \cdot \sigma_\Psi) \cdot \sigma'_\Psi = \tau \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$
3. $(e \cdot \sigma_\Psi) \cdot \sigma'_\Psi = e \cdot (\sigma_\Psi \cdot \sigma'_\Psi)$

By induction, and use of lemma B.94.

Lemma C.12 *If $\sigma_\Psi \subseteq \sigma'_\Psi$ then:*

1. *If $k \cdot \sigma_\Psi$ is defined, then $k \cdot \sigma_\Psi = k \cdot \sigma'_\Psi$*
2. *If $\tau \cdot \sigma_\Psi$ is defined, then $\tau \cdot \sigma_\Psi = \tau \cdot \sigma'_\Psi$*
3. *If $e \cdot \sigma_\Psi$ is defined, then $e \cdot \sigma_\Psi = e \cdot \sigma'_\Psi$*
4. *If $\Gamma \cdot \sigma_\Psi$ is defined, then $\Gamma \cdot \sigma_\Psi = \Gamma \cdot \sigma'_\Psi$*

Most are trivial based on induction and use of lemma B.92

Lemma C.13 1. *If $\Psi \vdash k$ wf and $\Psi' \vdash \sigma_\Psi : \Psi$ then $\Psi' \vdash k \cdot \sigma_\Psi$ wf.*
2. *If $\Psi;\ \Gamma \vdash \tau : k$ and $\Psi' \vdash \sigma_\Psi : \Psi$ then $\Psi';\ \Gamma \cdot \sigma_\Psi \vdash \tau \cdot \sigma_\Psi : k \cdot \sigma_\Psi$.*
3. *If $\Psi;\ \Sigma;\ \Gamma \vdash e : \tau$ and $\Psi' \vdash \sigma_\Psi : \Psi$ then $\Psi';\ \Sigma;\ \Gamma \cdot \sigma_\Psi \vdash e \cdot \sigma_\Psi : \tau \cdot \sigma_\Psi$.*

We only prove the interesting cases.

Part 1 **Case** $\dfrac{\vdash \Psi,\ K\ \text{wf} \qquad \Psi,\ K \vdash \lceil k\rceil_{|\Psi|,1}\ \text{wf}}{\Psi \vdash \Pi(K).k\ \text{wf}}\ \triangleright$

We use the induction hypothesis for $\Psi',\ K \cdot \sigma_\Psi \vdash (\sigma_\Psi,\ X_{|\Psi'|}) : \Psi,\ K$ and $\lceil k\rceil$ to get:
$\Psi',\ K \cdot \sigma_\Psi \vdash \lceil k\rceil_{|\Psi|,1} \cdot (\sigma_\Psi,\ X_{|\Psi'|})$ wf.
From lemma C.9 we have that $\lceil k\rceil_{|\Psi|,1} \cdot (\sigma_\Psi,\ X_{|\Psi'|}) = \lceil k \cdot \sigma_\Psi\rceil_{|\Psi'|,1}$.
Therefore by use of the same typing rule we have the desired result.

Part 2 **Case** $\dfrac{\Psi,\ K;\ \Gamma \vdash \lceil \tau\rceil_{|\Psi|,1} : k}{\Psi;\ \Gamma \vdash \lambda(K).\tau : \Pi(K).\ \lfloor k\rfloor_{|\Psi|,1}}\ \triangleright$

We use the induction hypothesis for $\Psi',\ K \vdash (\sigma_\Psi,\ X_{|\Psi'|}) : \Psi,\ K$ and $\lceil \tau\rceil$ to get, together with lemma C.9:
$\Psi',\ K \cdot \sigma_\Psi;\ \Gamma \cdot (\sigma_\Psi,\ X_{|\Psi'|}) \vdash \lceil \tau \cdot \sigma_\Psi\rceil_{|\Psi'|,1}$
By C.12 and the fact that $\Psi \vdash \Gamma$ wf, we have that $\Gamma \cdot (\sigma_\Psi,\ X_{|\Psi'|}) = \Gamma \cdot \sigma_\Psi$, so:
$\Psi',\ K \cdot \sigma_\Psi;\ \Gamma \cdot \sigma_\Psi \vdash \lceil \tau \cdot \sigma_\Psi\rceil_{|\Psi'|,1} : k \cdot (\sigma_\Psi,\ X_{|\Psi'|} \cdot \sigma_\Psi)$
By use of the same typing rule we get:
$\Psi';\ \Gamma \cdot \sigma_\Psi \vdash \lambda(K \cdot \sigma_\Psi).(\tau \cdot \sigma_\Psi) : \Pi(K \cdot \sigma_\Psi).(\lfloor k \cdot (\sigma_\Psi,\ X_{|\Psi'|})\rfloor_{|\Psi'|,1})$.
We have that $\lfloor k \cdot (\sigma_\Psi,\ X_{|\Psi'|})\rfloor_{|\Psi'|,1} = \lfloor k\rfloor_{|\Psi|,1} \cdot \sigma_\Psi$ by lemma C.10, so this is the desired result.

Case $\dfrac{\Psi;\,\Gamma \vdash \tau : \Pi(K).k \qquad \Psi \vdash T : K}{\Psi;\,\Gamma \vdash \tau\, T : \lceil k \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)}\ \triangleright$

By induction hypothesis we have:
$\Psi';\, \Gamma \cdot \sigma_\Psi \vdash \tau \cdot \sigma_\Psi : \Pi(K \cdot \sigma_\Psi).(k \cdot \sigma_\Psi)$
By use of B.97 for T we have:
$\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi$
By use of the same typing rule we get:
$\Psi';\, \Gamma \cdot \sigma_\Psi \vdash (\tau \cdot \sigma_\Psi)\ (T \cdot \sigma_\Psi) : \lceil k \cdot \sigma_\Psi \rceil_{|\Psi'|,1} \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi)$
Now we need to prove that $\lceil k \cdot \sigma_\Psi \rceil_{|\Psi'|,1} \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi) = (\lceil k \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)) \cdot \sigma_\Psi$.
From lemma C.9, we get that the left-hand side is equal to:
$(\lceil k \rceil_{|\Psi|,1} \cdot (\sigma_\Psi,\, X_{|\Psi'|})) \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi)$.
By application of lemma C.11 we get that it is further equal to:
$(\lceil k \rceil_{|\Psi|,1}) \cdot ((\sigma_\Psi,\, X_{|\Psi'|}) \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi))$.
By application of the same lemma to the right-hand side we have that it is equal to: $(\lceil k \rceil_{|\Psi|,1}) \cdot$
$((\mathsf{id}_{|\Psi|},\, T) \cdot \sigma_\Psi)$.
Thus we only need to prove that $(\sigma_\Psi,\, X_{|\Psi'|}) \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi) = (\mathsf{id}_{|\Psi|},\, T) \cdot \sigma_\Psi$.
We have that the left-hand side is equal to:
$\sigma_\Psi \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi),\, T \cdot \sigma_\Psi = \sigma_\Psi \cdot \mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi$ by lemma B.92.
Furthermore by lemma B.96 we have that $\sigma_\Psi \cdot \mathsf{id}_{\Psi'} = \sigma_\Psi$.
The right-hand side is equal to:
$\mathsf{id}_{|\Psi|} \cdot \sigma_\Psi,\, T \cdot \sigma_\Psi = \sigma_\Psi,\, T \cdot \sigma_\Psi$ due to lemma B.95.

Part 3 Most cases are proved as above, using the above lemmas. The most difficult case is the pattern matching construct.

$$\textbf{Case}\ \frac{\begin{array}{c}\Psi \vdash T : K \qquad \Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star \qquad \Psi \vdash \lceil \Psi'' \rceil_{|\Psi|}\ \mathrm{wf} \\ \Psi,\, \lceil \Psi'' \rceil_{|\Psi|} \vdash \lceil T' \rceil_{|\Psi|,|\Psi''|} : K \qquad \Psi,\, \lceil \Psi'' \rceil_{|\Psi|};\, \Sigma;\, \Gamma \vdash \lceil e' \rceil_{|\Psi|,|\Psi''|} : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, \lceil T' \rceil_{|\Psi|,|\Psi''|})\end{array}}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi''.T' \mapsto e') : (\lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)) + \mathsf{unit}}\ \triangleright$$

From $\Psi \vdash T : K$ and lemma B.97 we have:
$\Psi' \vdash T \cdot \sigma_\Psi : K \cdot \sigma_\Psi$
From $\Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star$, $\Psi \vdash \Gamma$ wf, part 2 and lemma C.9 we have:
$\Psi',\, K \cdot \sigma_\Psi;\, \Gamma \cdot \sigma_\Psi \vdash \lceil \tau \cdot \sigma_\Psi \rceil_{|\Psi'|,1} : \star$
From $\Psi \vdash \lceil \Psi'' \rceil_{|\Psi|}$ wf and lemmas B.98 and B.106 we have:
$\Psi' \vdash \lceil \Psi'' \cdot \sigma_\Psi \rceil_{|\Psi'|}$ wf
From $\Psi,\, \lceil \Psi'' \rceil_{|\Psi|};\, \Sigma;\, \Gamma \vdash \lceil e' \rceil_{|\Psi|,|\Psi''|} : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T')$, $\sigma'_\Psi = \sigma_\Psi,\, X_{|\Psi'|},\, \cdots,\, X_{|\Psi'|+|\Psi'' \cdot \sigma_\Psi|-1}$ and $\Psi',\, \lceil \Psi'' \cdot \sigma_\Psi \rceil_{|\Psi'|} \vdash$
$\sigma'_\Psi : (\Psi,\, \lceil \Psi'' \rceil_{|\Psi|})$, lemma B.97, lemma B.103, and lemma B.92 we have:
$\Psi',\, \lceil \Psi'' \cdot \sigma_\Psi \rceil_{|\Psi'|} \vdash \lceil T' \cdot \sigma_\Psi \rceil_{|\Psi'|,|\Psi'' \cdot \sigma_\Psi|} : K \cdot \sigma_\Psi$
Similarly for the same σ'_Ψ, and from $\Psi,\, \lceil \Psi'' \rceil_{|\Psi|};\, \Sigma;\, \Gamma \vdash \lceil e' \rceil_{|\Psi|,|\Psi''|} : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, \lceil T' \rceil_{|\Psi|,|\Psi''|})$, lemma C.9 and induction hypothesis, we get:
$\Psi',\, \lceil \Psi'' \cdot \sigma_\Psi \rceil_{|\Psi'|};\, \Sigma;\, \Gamma \cdot \sigma_\Psi \vdash \lceil e' \cdot \sigma_\Psi \rceil_{|\Psi'|,|\Psi'' \cdot \sigma_\Psi|} : (\lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, \lceil T' \rceil_{|\Psi|,|\Psi''|})) \cdot \sigma'_\Psi$.
Thus we only need to prove that $(\lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, \lceil T' \rceil_{|\Psi|,|\Psi''|})) \cdot \sigma'_\Psi = \lceil \tau \cdot \sigma_\Psi \rceil_{|\Psi'|,1} \cdot (\mathsf{id}_{\Psi'},\, \lceil T' \cdot \sigma_\Psi \rceil_{|\Psi'|,|\Psi'' \cdot \sigma_\Psi|})$.
In that case we will use the same typing rule to get the desired result, using a similar proof as this last step, to go from $\lceil \tau \cdot \sigma_\Psi \rceil_{|\Psi'|,1} \cdot (\mathsf{id}_{\Psi'},\, T \cdot \sigma_\Psi)$ to $(\lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)) \cdot \sigma_\Psi$.

So we now prove $(\lceil\tau\rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\ \lceil T'\rceil_{|\Psi|,|\Psi''|})) \cdot \sigma'_\Psi = \lceil\tau\cdot\sigma_\Psi\rceil_{|\Psi'|,1} \cdot (\mathsf{id}_{\Psi'},\ \lceil T'\cdot\sigma_\Psi\rceil_{|\Psi'|,|\Psi''\cdot\sigma_\Psi|})$:
By lemma C.9 and lemma B.103, we have that the right-hand side is equal to:
$(\lceil\tau\rceil_{|\Psi|,1} \cdot (\sigma_\Psi,\ X_{|\Psi'|})) \cdot (\mathsf{id}_{\Psi'},\ \lceil T'\rceil_{|\Psi|,|\Psi''|} \cdot \sigma'_\Psi)$.
By application of lemma C.11 we see that both sides are equal if $(\sigma_\Psi,\ X_{|\Psi'|}) \cdot (\mathsf{id}_{\Psi'},\ \lceil T'\rceil_{|\Psi|,|\Psi''|} \cdot \sigma'_\Psi) = (\mathsf{id}_\Psi,\ \lceil T'\rceil_{|\Psi|,|\Psi''|}) \cdot \sigma'_\Psi$.
The left-hand side of this is equal to $\sigma_\Psi \cdot (\mathsf{id}_{\Psi'},\ \lceil T'\rceil_{|\Psi|,|\Psi''|} \cdot \sigma'_\Psi),\ \lceil T'\rceil_{|\Psi|,|\Psi''|} \cdot \sigma'_\Psi$.
By lemma B.92 and B.96 we get that this is further equal to $\sigma_\Psi,\ \lceil T'\rceil_{|\Psi|,|\Psi''|} \cdot \sigma'_\Psi$.
The right-hand side is equal to $\mathsf{id}_\Psi \cdot \sigma'_\Psi,\ \lceil T'\rceil_{|\Psi|,|\Psi''|} \cdot \sigma'_\Psi$, which is equal to the above using lemmas B.92 and B.95.

Lemma C.14 *1.* $\left\lceil \lfloor k \rfloor^M_{N,K} \right\rceil^M_{N,K} = k$

2. $\left\lceil \lfloor \tau \rfloor^M_{N,K} \right\rceil^M_{N,K} = \tau$

3. $\left\lceil \lfloor e \rfloor^M_{N,K} \right\rceil^M_{N,K} = e$

Trivial by induction and use of lemma B.105.

Lemma C.15 (Substitution) *1. If* $\Psi,\ \Psi';\ \Gamma,\ \alpha' : k',\ \Gamma' \vdash \tau : k$ *and* $\Psi;\ \Gamma \vdash \tau' : k'$ *then* $\Psi,\ \Psi';\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash \tau[\tau'/\alpha'] : k$.

2. If $\Psi,\ \Psi';\ \Sigma\,\Gamma,\ \alpha' : k',\ \Gamma' \vdash e : \tau$ *and* $\Psi;\ \Gamma \vdash \tau' : k'$ *then* $\Psi,\ \Psi';\ \Sigma;\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash e[\tau'/\alpha'] : \tau[\tau'/\alpha']$.

3. If $\Psi,\ \Psi';\ \Sigma\,\Gamma,\ x' : \tau',\ \Gamma' \vdash e : \tau$ *and* $\Psi;\ \Sigma;\ \Gamma \vdash e' : \tau'$ *then* $\Psi,\ \Psi';\ \Sigma;\ \Gamma,\ \Gamma' \vdash e[e'/x'] : \tau$.

Easily proved by structural induction on the typing derivations.

Let us now proceed to prove the main preservation theorem.

Theorem C.16 (Preservation) *If* $\bullet;\ \Sigma;\ \bullet \vdash e : \tau,\ \mu \sim \Sigma,\ (\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$ *then there exists* Σ' *such that* $\Sigma \subseteq \Sigma',\ \mu' \sim \Sigma'$ *and* $\bullet;\ \Sigma';\ \bullet \vdash e' : \tau$.

Proceed by induction on the derivation of $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e\ )$. When we don't specify a different μ', we have that $\mu' = \mu$, with the desired properties obviously holding.

Case $\dfrac{(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )}{(\ \mu\ ,\ \mathcal{E}[e]\ ) \longrightarrow (\ \mu'\ ,\ \mathcal{E}[e']\ )}\ \triangleright$
By induction hypothesis for $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu'\ ,\ e'\ )$ we get a Σ' such that $\Sigma \subseteq \Sigma',\ \mu' \sim \Sigma'$ and $\bullet;\ \Sigma;\ \bullet \vdash e' : \tau$. By inversion of typing for $\mathcal{E}[e]$ and re-application of the same typing rule for $\mathcal{E}[e']$ we get that $\bullet;\ \Sigma';\ \bullet \vdash \mathcal{E}[e'] : \tau$.

Case $(\ \mu\ ,\ (\Lambda(K).e)\ T\ ) \longrightarrow (\ \mu\ ,\ \lceil e\rceil_{0,1} \cdot T\ )\ \triangleright$
By inversion of typing we get:
$\bullet;\ \Sigma;\ \bullet \vdash \Lambda(K).e : \Pi(K).\tau'$
$\bullet \vdash T : K$
$\tau = \lceil\tau'\rceil_{0,1} \cdot T$
By further typing inversion for $\Lambda(K).e$ we get:
$\bullet,\ K;\ \Sigma;\ \bullet \vdash \lceil e\rceil_{0,1} : \tau''$
$\tau' = \lfloor\tau''\rfloor_{0,1}$
For $\sigma_\Psi = \bullet,\ T$ we have $\bullet \vdash (\bullet,\ T) : (\bullet,\ K)$ trivially from the above.
By lemma C.13 for σ_Ψ we get that:

$\bullet; \Sigma; \bullet \vdash \lceil e \rceil_{0,1} \cdot T : \tau'' \cdot T$.
Now it remains to show that $\tau'' \cdot T = \left\lceil \lfloor \tau'' \rfloor_{0,1} \right\rceil_{0,1} \cdot T$, which is proved by C.14.

Case $(\ \mu\ ,\ \mathsf{unpack}\ \langle T,\ \tau \rangle\ v\ (.)x.(e')\ ) \longrightarrow (\ \mu\ ,\ (\lceil e' \rceil_{0,1} \cdot T)[v/x]\ )\ \triangleright$
By inversion of typing we get:
$\bullet; \Sigma; \bullet \vdash \langle T,\ \tau'' \rangle\ v : \Sigma(K).\tau'$
$\bullet,\ K; \Sigma; x : \lceil \tau' \rceil_{0,1} \vdash \lceil e' \rceil_{0,1} : \tau$
$\bullet;\ \bullet \vdash \tau : \star$
By further typing inversion for $\langle T,\ \tau'' \rangle\ v$ we get:
$\tau'' = \tau'$
$\bullet \vdash T : K$
$\bullet,\ K; \bullet \vdash \lceil \tau' \rceil_{0,1} : \star$
$\bullet; \Sigma; \bullet \vdash v : \lceil \tau' \rceil_{0,1} \cdot (T)$
First by lemma C.13 for e', $\sigma_\Psi = T$ we get:
$\bullet; \Sigma; x : \lceil \tau' \rceil_{0,1} \cdot T \vdash \lceil e' \rceil_{0,1} \cdot T : \tau \cdot T$.
Second by lemma C.12 for τ we get that $\tau \cdot T = \tau$.
Thus $\bullet; \Sigma; x : \lceil \tau' \rceil_{0,1} \cdot T \vdash \lceil e' \rceil_{0,1} \cdot T : \tau$.
Furthermore by lemma C.15 for $[v/x]$ we get that $\bullet; \Sigma; \bullet \vdash (\lceil e' \rceil_{0,1} \cdot T)[v/x] : \tau$, which is the desired.

Case $(\ \mu\ ,\ (\lambda x : \tau.e)\ v\ ) \longrightarrow (\ \mu\ ,\ e[v/x]\ )\ \triangleright$
By inversion of typing we get:
$\bullet; \Sigma; \bullet \vdash \lambda x : \tau.e : \tau' \to \tau$
$\bullet; \Sigma; \bullet \vdash v : \tau'$
By further typing inversion for $\lambda x : \tau.e$ we get:
$\bullet; \Sigma; x : \tau \vdash e : \tau$
By lemma C.15 for $[v/x]$ we get:
$\bullet; \Sigma; \bullet \vdash e[v/x] : \tau$, which is the desired.

Case $(\ \mu\ ,\ \mathsf{proj}_i(v_1,\ v_2)\ ) \longrightarrow (\ \mu\ ,\ v_i\ )\ \triangleright$
By typing inversion we get:
$\bullet; \Sigma; \bullet \vdash (v_1,\ v_2) : \tau_1 \times \tau_2$
$\tau = \tau_i$
By further inversion for $(v_1,\ v_2)$ we have:
$\bullet; \Sigma; \bullet \vdash v_i : \tau_i$, which is the desired.

Case $(\ \mu\ ,\ \mathsf{case}(\mathsf{inj}_i\ v, x.e_1, x.e_2)\ ) \longrightarrow (\ \mu\ ,\ e_i[v/x]\ )\ \triangleright$
By typing inversion we get:
$\bullet; \Sigma; \bullet \vdash v : \tau_i$
$\bullet; \Sigma; x : \tau_i \vdash e_i : \tau$
Using the lemma C.15 for [v/x] we get:
$\bullet; \Sigma; \bullet \vdash e_i[v/x] : \tau$

Case $(\ \mu\ ,\ \mathsf{unfold}\ (\mathsf{fold}\ v)\ ) \longrightarrow (\ \mu\ ,\ v\ )\ \triangleright$
By inversion we get: $\bullet; \bullet \vdash \mu\alpha : k.\tau' : k$
$\bullet; \Sigma; \bullet \vdash \mathsf{fold}\ v : (\mu\alpha : k.\tau')\ \tau_1\ \tau_2\ \cdots\ \tau_n$
$\tau = \tau'[\mu\alpha : k.\tau']\ \tau_1\ \tau_2\ \cdots\ \tau_n$
By further typing inversion for fold v:

$\bullet$; Σ; $\bullet \vdash v : \tau'[\mu\alpha : k.\tau/\alpha]\ \tau_1\ \tau_2\ \cdots\ \tau_n$
Which is the desired.

Case $\dfrac{l \mapsto \_ \notin \mu}{(\ \mu\ ,\ \mathsf{ref}\ v\ ) \longrightarrow (\ (\mu,\ l \mapsto v)\ ,\ l\ )}\ \triangleright$
By typing inversion we get: $\bullet$; Σ; $\bullet \vdash v : \tau$
For $\Sigma' = \Sigma$, $l : \tau$ and $\mu' = \mu,\ l \mapsto v$ we have that $\mu' \sim \Sigma'$ and $\bullet$; Σ'; $\bullet \vdash l : \mathsf{ref}\ \tau$.

Case $\dfrac{l \mapsto \_ \in \mu}{(\ \mu\ ,\ l := v\ ) \longrightarrow (\ \mu[l \mapsto v]\ ,\ ()\ )}\ \triangleright$
By typing inversion get:
$\bullet$; Σ; $\bullet \vdash l : \mathsf{ref}\ \tau$
$\bullet$; Σ; $\bullet \vdash v : \tau$
Thus for $\mu' = \mu[l \mapsto v]$ we have that $\mu' \sim \Sigma$ and $\bullet$; Σ; $\bullet \vdash () : \mathsf{unit}$.

Case $\dfrac{l \mapsto v \in \mu}{(\ \mu\ ,\ !l\ ) \longrightarrow (\ \mu\ ,\ v\ )}\ \triangleright$
By typing inversion get: $\bullet$; Σ; $\bullet \vdash l : \mathsf{ref}\ \tau$
By inversion of $\mu \sim \Sigma$ get:
$\bullet$; Σ; $\bullet \vdash v : \tau$, which is the desired.

Case $(\ \mu\ ,\ (\Lambda\alpha : k.e)\ \tau''\ ) \longrightarrow (\ \mu\ ,\ e[\tau''/\alpha]\ )\ \triangleright$
By typing inversion we get:
$\bullet$; Σ; $\bullet \vdash \Lambda\alpha : k.e : \Pi\alpha : k.\tau'$
$\bullet$; $\bullet \vdash \tau'' : k$
$\tau = \tau'[\tau''/\alpha]$
By further typing inversion for $\Lambda\alpha : k.e$ we get:
$\bullet$; Σ; $\alpha : k \vdash e : \tau'$
Using the lemma C.15 for $[\tau''/\alpha]$ we get:
$\bullet$; Σ; $\bullet \vdash e[\tau''/\alpha] : \tau'[\tau''/\alpha]$, which is the desired.

Case $(\ \mu\ ,\ \mathsf{fix}\ x : \tau.e\ ) \longrightarrow (\ \mu\ ,\ e[\mathsf{fix}\ x : \tau.e/x]\ )\ \triangleright$
By typing inversion get:
$\bullet$; Σ; $x : \tau \vdash e : \tau$
By application of the lemma C.15 for $[\mathsf{fix}\ x : \tau.e/x]$ we get:
$\bullet$; Σ; $\bullet \vdash e[\mathsf{fix}\ x : \tau.e/x] : \tau$

Case $\dfrac{\exists \sigma_\Psi.(\bullet \vdash \sigma_\Psi : \lceil \Psi' \rceil_0 \quad \wedge \quad \lceil T' \rceil_{0,|\Psi'|} \cdot \sigma_\Psi = T)}{(\ \mu\ ,\ \mathsf{unify}\ T\ \mathsf{return}\ (.\tau')\ \mathsf{with}\ (\Psi'.T' \mapsto e')\ ) \longrightarrow (\ \mu\ ,\ \mathsf{inj}_1\ (\lceil e' \rceil_{0,|\Psi'|} \cdot \sigma_\Psi)\ )}\ \triangleright$
By inversion of typing we get:
$\bullet \vdash T : K$
$\bullet$, K; Σ; $\bullet \vdash \lceil \tau' \rceil_{0,1} : \star$
$\bullet \vdash \lceil \Psi' \rceil_{|\Psi|}$ wf
$\lceil \Psi' \rceil_0 \vdash \lceil T' \rceil_{0,|\Psi'|} : K$
$\lceil \Psi' \rceil_0$; Σ; $\bullet \vdash \lceil e' \rceil_{0,|\Psi'|} : \lceil \tau' \rceil_{0,1} \cdot (T')$
$\tau = (\lceil \tau' \rceil_{0,1} \cdot T) + \mathsf{unit}$
By application of lemma C.15 for σ_Ψ and $\lceil e' \rceil_{0,|\Psi'|}$ we get:

$\bullet;\ \Sigma;\ \bullet \vdash \lceil e' \rceil_{0,|\Psi'|} \cdot \sigma_\Psi : (\lceil \tau' \rceil_{0,1} \cdot T') \cdot \sigma_\Psi$.
All we now need to prove is that $\lceil \tau' \rceil_{0,1} \cdot T = (\lceil \tau' \rceil_{0,1} \cdot T') \cdot \sigma_\Psi$.
Using the lemma C.11 we get that:
$(\lceil \tau' \rceil_{0,1} \cdot T') \cdot \sigma_\Psi = \lceil \tau' \rceil_{0,1} \cdot (T' \cdot \sigma_\Psi) = \lceil \tau' \rceil_{0,1} \cdot T$
It is now easy to complete the desired result using the typing rule for inj_1.

Case $\dfrac{\nexists \sigma_\Psi.(\bullet \vdash \sigma_\Psi : \lceil \Psi \rceil_0 \quad \wedge \quad \lceil T' \rceil_{0,|\Psi|} \cdot \sigma_\Psi = T)}{(\ \mu\ ,\ \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e')\ ) \longrightarrow (\ \mu\ ,\ \mathsf{inj}_2\ ()\ )}\ \triangleright$
Trivial by application of the typing rule for inj_2.

Lemma C.17 (Canonical forms) *If* $\bullet;\ \Sigma;\ \bullet \vdash v : \tau$ *then*

1. *If* $\tau = \Pi(K).\tau'$, *then* $\exists e$ *such that* $v = \Lambda(K).e$.
2. *If* $\tau = \Sigma(K).\tau'$, *then* $\exists T, v'$ *such that* $v = \mathsf{pack}\ T\ \mathsf{return}\ (.\tau'')\ \mathsf{with}\ v'$ *with* $\tau' =_\beta \tau''$.
3. *If* $\tau = \mathsf{unit}$, *then* $v = ()$.
4. *If* $\tau = \tau_1 \to \tau_2$, *then* $\exists e$ *such that* $v = \lambda x : \tau_1.e$.
5. *If* $\tau = \tau_1 \times \tau_2$, *then* $\exists v_1, v_2$ *such that* $v = (v_1,\ v_2)$.
6. *If* $\tau = \tau_1 + \tau_2$, *then* $\exists v'$ *such that either* $v = \mathsf{inj}_1\ v'$ *or* $v = \mathsf{inj}_2\ v'$.
7. *If* $\tau = (\mu\alpha : k.\tau')\ \tau_1\ \tau_2\ \cdots\ \tau_n$, *then* $\exists v'$ *such that* $v = \mathsf{fold}\ v'$.
8. *If* $\tau = \mathsf{ref}\ \tau'$, *then* $\exists l$ *such that* $v = l$.
9. *If* $\tau = \Lambda\alpha : k.\tau'$, *then* $\exists e$ *such that* $v = \Lambda\alpha : k.e$.

Directly by typing inversion.

Theorem C.18 (Progress) *If* $\bullet;\ \Sigma;\ \bullet \vdash e : \tau$ *and* $\mu \sim \Sigma$, *then either* $\mu,\ e \longrightarrow \mathsf{error}$, *or* e *is a value* v, *or there exist* μ' *and* e' *such that* $\mu,\ e \longrightarrow \mu',\ e'$.

We proceed by induction on the typing derivation for e. We do not consider cases where $e = v$ (since the theorem is trivial in that case), or where $e = \mathcal{E}[e']$ with $e \not= v$. In that case, by typing inversion we can get that e' is well-typed under the empty context, so by induction hypothesis we can either prove that $\mu,\ e \longrightarrow \mathsf{error}$, or there exist μ', e'' such that $\mu,\ \mathcal{E}[e] \longrightarrow \mu',\ \mathcal{E}[e'']$ by the environment closure small-step rule. Thus we only consider cases where $e = \mathcal{E}[v]$, or where e cannot be further decomposed into $\mathcal{E}[e']$ with $\mathcal{E} \not= \bullet$. Last, when we don't mention a specific μ', we have that $\mu' = \mu$ with the desired properties obviously holding.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \Pi(K).\tau \qquad \bullet \vdash T : K}{\bullet;\ \Sigma;\ \bullet \vdash v\ T : \lceil \tau \rceil_{0,1} \cdot (T)}\ \triangleright$
By use of the canonical forms lemma C.17, we get that $v = \Lambda(K).e$.
By typing inversion we get that $K;\ \Sigma;\ \bullet \vdash \lceil e \rceil_{0,1} : \tau'$.
So applying the appropriate operational semantics rule we get an $e' = \lceil e \rceil_{0,1} \cdot T$ such that $(\ \mu\ ,\ e\ ) \longrightarrow (\ \mu\ ,\ e'\ )$.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \Sigma(K).\tau \qquad \bullet,\ K,\ \Sigma;\ \bullet,\ x : \lceil \tau \rceil_{0,1} \vdash \lceil e' \rceil_{0,1} : \tau' \qquad \bullet;\ \bullet \vdash \tau' : \star}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{unpack}\ v\ (.)x.(e') : \tau'}\ \triangleright$
By use of the canonical forms lemma C.17, we get that $v = \mathsf{pack}\ T\ \mathsf{return}\ (.\tau'')\ \mathsf{with}\ v'$.
Furthermore we have that $\lceil e' \rceil_{0,1}$ is well-defined, so such will be $\lceil e' \rceil_{0,1} \cdot T$ too.
Thus the relevant operational semantics rule applies.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \tau \to \tau' \qquad \bullet;\ \Sigma;\ \bullet \vdash e' : \tau}{\bullet;\ \Sigma;\ \bullet \vdash v\ e' : \tau'}\ \triangleright$

From canonical forms, we have that $v = \lambda x : \tau''.e''$, so the relevant step rule applies.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash e : \tau_1 \times \tau_2 \qquad i = 1 \text{ or } 2}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{proj}_i\ e : \tau_i}\ \triangleright$

From canonical forms, we have that $v = (v_1,\ v_2)$, so using the relevant step rule for proj_i we get that $(\ \mu\ ,\ \mathsf{proj}_i\ e\ ) \longrightarrow (\ \mu\ ,\ v_i\ )$.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \tau_1 + \tau_2 \qquad \bullet;\ \Sigma;\ \bullet,\ x : \tau_1 \vdash e_1 : \tau \qquad \bullet;\ \Sigma;\ \bullet,\ x : \tau_2 \vdash e_2 : \tau}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{case}(v, x.e_1, x.e_2) : \tau}\ \triangleright$

From canonical forms, we have that either $v = \mathsf{inj}_1\ v'$ or $v = \mathsf{inj}_2\ v'$; in each case a step rule applies to give an appropriate e'.

Case $\dfrac{\bullet;\ \bullet \vdash \mu\alpha : k.\tau : k \qquad \bullet;\ \Sigma;\ \bullet \vdash v : (\mu\alpha : k.\tau)\ \tau_1\ \tau_2\ \cdots\ \tau_n}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{unfold}\ v : \tau[\mu\alpha : k.\tau/\alpha]\ \tau_1\ \tau_2\ \cdots\ \tau_n}\ \triangleright$

From canonical forms, we get that $v = \mathsf{fold}\ v'$, so the relevant step rule trivially applies.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \tau}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{ref}\ v : \mathsf{ref}\ \tau}\ \triangleright$

Assuming an infinite heap, we can find a l such that $l \notin \mu$, and construct $\mu' = \mu,\ l \mapsto v$. Thus the relevant step rule applies giving $e' = l$.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \mathsf{ref}\ \tau}{\bullet;\ \Sigma;\ \bullet \vdash !v : \tau}\ \triangleright$

From canonical forms, we get that $v = l$. By typing inversion, we get that $(l : \tau) \in \Sigma$.
From $\mu \sim \Sigma$, we get that there exists v' such that $(l \mapsto v') \in \mu$.
Thus the relevant step rule applies and gives $e' = v'$.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet \vdash v : \Pi\alpha : k.\tau' \qquad \bullet;\ \bullet \vdash \tau : k}{\bullet;\ \Sigma;\ \bullet \vdash v\ \tau : \tau'[\tau/\alpha]}\ \triangleright$

From canonical forms, we get that $v = \Lambda\alpha : k.e$. The relevant step rule trivially applies to give $e' = e[\tau/\alpha]$.

Case $\dfrac{\bullet;\ \Sigma;\ \bullet,\ x : \tau \vdash e : \tau}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{fix}\ x : \tau.e : \tau}\ \triangleright$

Trivially we have that the relevant step rule applies giving $e' = e[\mathsf{fix}\ x : \tau.e/x]$.

Case $\dfrac{\begin{array}{c}\bullet \vdash T : K \qquad \bullet,\ K;\ \bullet \vdash \lceil\tau\rceil_{0,1} : \star \\ \bullet \vdash \left\lceil\Psi'\right\rceil_0\ \mathrm{wf} \qquad \bullet,\ \left\lceil\Psi'\right\rceil_0 \vdash \left\lceil T'\right\rceil_{0,|\Psi'|} : K \qquad \bullet,\ \left\lceil\Psi'\right\rceil_0;\ \Sigma;\ \bullet \vdash \left\lceil e'\right\rceil_{0,|\Psi'|} : \lceil\tau\rceil_{0,1} \cdot \left(\left\lceil T'\right\rceil_{0,|\Psi'|}\right)\end{array}}{\bullet;\ \Sigma;\ \bullet \vdash \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi'.T' \mapsto e') : (\lceil\tau\rceil_{0,1} \cdot (T)) + \mathsf{unit}}\ \triangleright$

We have non-determinism here in the semantics, which we will fix in the next section, giving more precise semantics to the patterns and unification procedure. In either case, we split cases on whether an σ_Ψ with the desired properties exists or not, and use the appropriate step rule to get an e' in each case.

D. Typing and unification for patterns

D.1 Adjusting computational language typing

First, we will define two new notions: one is a stricter typing for patterns, allowing only certain forms to be used; the second is relevant typing for patterns, making sure that all declared unification variables are actually used somewhere inside the pattern. Together they are supposed to make sure that unification is possible using a decidable deterministic algorithm; so there is only one unifying substitution, or there is none.

We change the pattern matching typing rule for the computational language as follows:

$$\frac{\begin{array}{c}\Psi \vdash T : K \qquad \Psi,\, K;\, \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star \qquad \Psi \vdash_p \lceil \Psi' \rceil_{|\Psi|}\ \mathrm{wf} \\ \Psi,\, \lceil \Psi' \rceil_{|\Psi|} \vdash_p \lceil T' \rceil_{|\Psi|,|\Psi'|} : K \qquad \mathsf{relevant}\left(\Psi,\, \lceil \Psi' \rceil_{|\Psi|} \vdash_p \lceil T' \rceil_{|\Psi|,|\Psi'|} : K\right) = \widehat{\Psi},\, \lceil \Psi' \rceil_{|\Psi|} \\ \Psi,\, \lceil \Psi' \rceil_{|\Psi|};\, \Sigma;\, \Gamma \vdash \lceil e' \rceil_{|\Psi|,|\Psi'|} : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, \lceil T' \rceil_{|\Psi|,|\Psi'|})\end{array}}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi'.T' \mapsto e') : (\lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\, T)) + \mathsf{unit}}$$

Then we define the stricter typing for patterns $\vdash_p$. This will be entirely identical to normal typing, but will disallow forms that would lead to non-determinism (e.g. context unification variables allowed anywhere inside a pattern).

Then we define the notion of relevancy for extension variables. For a judgement $\Psi; \mathcal{J}$, $\mathsf{relevant}\,(\Psi; \mathcal{J}) = \widehat{\Psi}$, where $\widehat{\Psi}$ is a partial context, containing only the extension variables that actually get used. We will show that functions used during typing and evaluation commute with this function.

Then, we prove that either a unique unification exists for a pair of a pattern and a term, yielding a partial substitution for the relevant variables, or that no such unification exists. From this proof we derive an algorithm for unification.

D.2 Strict typing for patterns

Definition D.1 (Pattern typing) *We will adapt the typing rules for extended terms T, to show which of those terms are accepted as valid patterns. We assume that the Ψ is split into two parts, Ψ, Ψ_u, where Ψ_u contains only newly-introduced unification variables just for the purpose of type-checking the current pattern and branch.*

$\boxed{\Psi \vdash_p \Psi_u\ \mathrm{wf}}$

$$\frac{}{\Psi \vdash_p \bullet\ \mathrm{wf}} \qquad \frac{\Psi \vdash_p \Psi_u\ \mathrm{wf} \qquad \Psi,\, \Psi_u \vdash_p [\Phi]t : [\Phi]s}{\Psi \vdash_p (\Psi_u,\, [\Phi]t)\ \mathrm{wf}} \qquad \frac{\Psi \vdash_p \Psi_u\ \mathrm{wf} \qquad \Psi,\, \Psi_u \vdash_p \Phi\ \mathrm{wf}}{\Psi \vdash_p (\Psi_u,\, [\Phi]\,\mathsf{ctx})\ \mathrm{wf}}$$

$\boxed{\Psi,\, \Psi_u \vdash_p T : K}$

$$\frac{\Psi,\, \Psi_u;\, \Phi \vdash_p t : t' \qquad \Psi,\, \Psi_u;\, \Phi \vdash t' : s}{\Psi,\, \Psi_u \vdash_p [\Phi]t : [\Phi]t'} \qquad \frac{\Psi,\, \Psi_u \vdash_p \Phi,\, \Phi'\ \mathrm{wf}}{\Psi,\, \Psi_u \vdash_p [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}}$$

$\boxed{\Psi,\, \Psi_u \vdash_p \Phi\ \mathrm{wf}}$

$$\frac{}{\Psi,\, \Psi_u \vdash_p \bullet\ \mathrm{wf}} \qquad \frac{\Psi,\, \Psi_u \vdash_p \Phi\ \mathrm{wf} \qquad \Psi,\, \Psi_u;\, \Phi \vdash_p t : s}{\Psi,\, \Psi_u \vdash_p (\Phi,\, t)\ \mathrm{wf}}$$

$$\frac{\Psi,\, \Psi_u \vdash_p \Phi\ \mathrm{wf} \qquad (\Psi,\, \Psi_u).i = [\Phi]\,\mathsf{ctx} \qquad i < |\Psi|}{\Psi,\, \Psi_u \vdash_p (\Phi,\, X_i)\ \mathrm{wf}} \qquad \frac{\Psi \vdash_p \Phi\ \mathrm{wf} \qquad (\Psi,\, \Psi_u).i = [\Phi]\,\mathsf{ctx} \qquad i \geq |\Psi|}{\Psi,\, \Psi_u \vdash_p \Phi,\, X_i\ \mathrm{wf}}$$

$\boxed{\Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi'}$

$$\frac{}{\Psi,\ \Psi_u;\ \Phi \vdash_p \bullet : \bullet} \qquad \frac{\Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi' \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t : t' \cdot \sigma}{\Psi,\ \Psi_u;\ \Phi \vdash_p (\sigma,\ t) : (\Phi',\ t')}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi' \qquad (\Psi,\ \Psi_u).i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\ X_i \subseteq \Phi}{\Psi,\ \Psi_u;\ \Phi \vdash_p (\sigma,\ \mathbf{id}(X_i)) : (\Phi',\ X_i)}$$

$\boxed{\Psi,\ \Psi_u;\ \Phi \vdash_p t : t'}$

$$\frac{c : t \in \Sigma}{\Psi,\ \Psi_u;\ \Phi \vdash_p c : t} \qquad \frac{\Phi.\mathbf{I} = t}{\Psi,\ \Psi_u;\ \Phi \vdash_p f_{\mathbf{I}} : t} \qquad \frac{(s,s') \in \mathcal{A}}{\Psi,\ \Psi_u;\ \Phi \vdash_p s : s'}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1 : s \qquad \Psi,\ \Psi_u;\ \Phi,\ t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s,s',s'') \in \mathcal{R}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \Pi(t_1).t_2 : s''}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1 : s \qquad \Psi,\ \Psi_u;\ \Phi,\ t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|,.} : s'}{\Psi,\ \Psi_u;\ \Phi \vdash_p \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|,.}}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1 : \Pi(t).t' \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t_2 : t}{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1\ t_2 : \lceil t' \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1 : t \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t_2 : t \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t : \mathsf{Type}}{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1 = t_2 : \mathsf{Prop}}$$

$$\frac{(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']t' \qquad i < |\Psi| \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi'}{\Psi,\ \Psi_u;\ \Phi \vdash_p X_i/\sigma : t' \cdot \sigma}$$

$$\frac{(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']t' \qquad i \geq |\Psi| \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi' \qquad \Phi' \subseteq \Phi \qquad \sigma = \mathsf{id}_{\Phi'}}{\Psi,\ \Psi_u;\ \Phi \vdash_p X_i/\sigma : t' \cdot \sigma}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t : t_1 \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t_1 : \mathsf{Prop} \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t' : t_1 = t_2}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{conv}\ t\ t' : t_2}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_1 : t \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t_1 = t_1 : \mathsf{Prop}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{refl}\ t_1 : t_1 = t_1} \qquad \frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_a : t_1 = t_2}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{symm}\ t_a : t_2 = t_1}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p t_a : t_1 = t_2 \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t_b : t_2 = t_3}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{trans}\ t_a\ t_b : t_1 = t_3}$$

$$\frac{\begin{array}{c}\Psi,\ \Psi_u;\ \Phi \vdash_p t_a : M_1 = M_2 \\ \Psi,\ \Psi_u;\ \Phi \vdash_p M_1 : A \to B \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p t_b : N_1 = N_2 \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p N_1 : A\end{array}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{congapp}\ t_a\ t_b : M_1\ N_1 = M_2\ N_2}$$

$$\frac{\begin{array}{c}\Psi,\ \Psi_u;\ \Phi \vdash_p t_a : A_1 = A_2 \\ \Psi,\ \Psi_u;\ \Phi,A_1 \vdash_p \lceil t_b \rceil : B_1 = B_2 \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p A_1 : \mathsf{Prop} \qquad \Psi,\ \Psi_u;\ \Phi,A_1 \vdash_p \lceil B_1 \rceil : \mathsf{Prop}\end{array}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{congimpl}\ t_a\ (\lambda(A_1).t_b) : \Pi(A_1).\ \lfloor B_1 \rfloor = \Pi(A_2).\ \lfloor B_2 \rfloor}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi,\ A \vdash_p \lceil t_b \rceil : B = B' \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p \Pi(A).\ \lfloor B \rfloor = \Pi(A).\ \lfloor B' \rfloor : \mathsf{Prop}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{congpi}\ (\lambda(A).t_b) : \Pi(A).\ \lfloor B \rfloor = \Pi(A).\ \lfloor B' \rfloor}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi,\ A \vdash_p \lceil t_b \rceil : B_1 = B_2 \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p \lambda(A).\ \lfloor B_1 \rfloor = \lambda(A).\ \lfloor B_2 \rfloor : \mathsf{Prop}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{conglam}\ (\lambda(A).t_b) : \lambda(A).\ \lfloor B_1 \rfloor = \lambda(A).\ \lfloor B_2 \rfloor}$$

$$\frac{\Psi,\ \Psi_u;\ \Phi \vdash_p \lambda(A).M : A \to B \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p N : A \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p A \to B : \mathsf{Type}}{\Psi,\ \Psi_u;\ \Phi \vdash_p \mathsf{beta}\ (\lambda(A).M)\ N : (\lambda(A).M)\ N = \lceil M \rceil \cdot (\mathsf{id}_\Phi, N)}$$

Now we need to prove that all theorems that had to do with these typing judgements still hold. In most cases, this holds entirely trivially, since the $\vdash_p$ judgements are exactly the same as the $\vdash$ judgements, with some extra restrictions as side-conditions. The only theorems that we need to reprove are the ones that require special care in exactly those rules that now have side-conditions. As these rules all have to do just with the use of extension variables, we understand that the theorems that we need to adapt are the extension substitution lemmas. Their statements need to be adjusted to account for part of the substitution corresponding to the Ψ part, and part of it corresponding to the Ψ_u part (both in the source and target extension contexts of the substitution). Though we do not provide the details here, the main argument why these continue to hold is the following: we never substitute variables from Ψ_u with anything other than the same variable in a context that includes the same Ψ_u. Thus the side-conditions will continue to hold.

Theorem D.2 (Extension of lemma B.97) *If* Ψ', $\Psi_u \cdot \sigma_\Psi \vdash (\sigma_\Psi, X_{|\Psi'|}, \cdots, X_{|\Psi'|+|\Psi_u|}) : (\Psi,\ \Psi_u)$ *then:*

1. *If* $\Psi,\ \Psi_u;\ \Phi \vdash_p t : t'$ *then* $\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma_\Psi \vdash_p t \cdot \sigma_\Psi : t' \cdot \sigma_\Psi$.
2. *If* $\Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi'$ *then* $\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma_\Psi \vdash_p \sigma \cdot \sigma_\Psi : \Phi' \cdot \sigma_\Psi$.
3. *If* $\Psi,\ \Psi_u \vdash_p \Phi$ *wf then* $\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash_p \Phi \cdot \sigma_\Psi$ *wf.*
4. *If* $\Psi,\ \Psi_u \vdash_p T : K$ *then* $\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash_p T \cdot \sigma_\Psi : K \cdot \sigma_\Psi$.

In all cases proceed entirely similarly as before. The only special cases that need to be accounted for are the ones that have to do with restrictions on variables coming out of Ψ_u.

Case $\dfrac{\Psi \vdash_p \Phi\ \mathsf{wf} \qquad (\Psi,\ \Psi_u).i = [\Phi]\,\mathsf{ctx} \qquad i \geq |\Psi|}{\Psi,\ \Psi_u \vdash_p \Phi,\ X_i\ \mathsf{wf}}\ \triangleright$

We need to prove that $\Psi',\ \Psi'_u \vdash_p \Phi \cdot \sigma'_\Psi,\ X_i \cdot \sigma'_\Psi$ wf, where $\sigma'_\Psi = \sigma_\Psi,\ X_{|\Psi'|},\ \cdots,\ X_{|\Psi'|+|\Psi_u|}$. By induction hypothesis for $\Psi_u = \bullet$ we get that $\Psi \vdash_p \Phi \cdot \sigma_\Psi$ wf, and because of lemma B.92 we get that also $\Psi \vdash_p \Phi \cdot \sigma'_\Psi$ wf.
Also, we have that $X_i \cdot \sigma'_\Psi = X_{i-|\Psi|+|\Psi'|}$.
We have that $(\Psi',\ \Psi'_u).i - |\Psi| + |\Psi'| = [\Phi \cdot \sigma'_\Psi]\,\mathsf{ctx}$.
Last, since $i \geq |\Psi|$, we also have that $i - |\Psi| + |\Psi'| \geq |\Psi'|$.
Thus by the use of the same typing rule, we arrive at the desired.

Case $\dfrac{(\Psi,\ \Psi_u).i = T \qquad T = [\Phi']t' \qquad i \geq |\Psi| \qquad \Psi,\ \Psi_u;\ \Phi \vdash_p \sigma : \Phi' \qquad \sigma = \mathsf{id}_{\Phi'}}{\Psi,\ \Psi_u;\ \Phi \vdash_p X_i/\sigma : t' \cdot \sigma}\ \triangleright$

Similarly as above. Furthermore, we need to show that $\mathsf{id}_{\Phi'} \cdot \sigma'_\Psi = \mathsf{id}_{\Phi' \cdot \sigma'_\Psi}$, which is simple to prove by induction on Φ'.

Lemma D.3 (Extension of lemma B.98) *If* $\Psi \vdash_p \Psi_u$ *wf then* $\Psi \vdash_p \Psi_u \cdot \sigma_\Psi$ *wf.*

Similarly to lemma B.98 and use of the above lemma.

D.3 Relevant typing

We will proceed to define a notion of partial contexts: these are extension contexts where certain elements are unspecified. It is presumed that in the judgements that they appear, only specified elements are relevant; the judgements do not depend on the other elements at all (save for them being well-formed). We will use this notion in order to make sure that all unification variables introduced during pattern matching are relevant. Otherwise, the irrelevant variables could be substituted by arbitrary terms, resulting in the existence of an infinite number of valid unification substitutions.

Definition D.4 *The syntax for partial contexts is defined as follows.*

$\widehat{\Psi} ::= \bullet \mid \widehat{\Psi},\, K \mid \widehat{\Psi},\, ?$

Definition D.5 *Well-formedness for partial contexts is defined as follows.*

$\boxed{\vdash \widehat{\Psi}\ \mathrm{wf}}$

$$\frac{}{\vdash \bullet\ \mathrm{wf}} \qquad \frac{\vdash \widehat{\Psi}\ \mathrm{wf} \qquad \widehat{\Psi} \vdash [\Phi]t : [\Phi]s}{\vdash (\widehat{\Psi},\, [\Phi]t)\ \mathrm{wf}} \qquad \frac{\vdash \widehat{\Psi}\ \mathrm{wf} \qquad \widehat{\Psi} \vdash \Phi\ \mathrm{wf}}{\vdash (\widehat{\Psi},\, [\Phi]\,\mathsf{ctx})\ \mathrm{wf}} \qquad \frac{\vdash \widehat{\Psi}\ \mathrm{wf}}{\vdash (\widehat{\Psi},\, ?)\ \mathrm{wf}}$$

Other than the above change in the well-formedness definition, partial contexts are used with entirely the same definitions as before. This means that if a typing judgement like $\widehat{\Psi};\ \Phi \vdash t : t'$ tries to access the i-th metavariable, this metavariable should be specified in $\widehat{\Psi}$ rather than being the unspecified element $?$ – because the side-condition $\widehat{\Psi}.i = K$ would otherwise be violated.

We proceed to define a judgement that extracts the relevant extension variables out of typing judgements that use a concrete context Ψ, yielding a partial context $\widehat{\Psi}$. We first need a couple of definitions.

Definition D.6 *The fully-unspecified partial context is defined as follows.*

$\boxed{\mathsf{unspec}_\Psi}$

$$\begin{array}{lcl} \mathsf{unspec}_\bullet & = & \bullet \\ \mathsf{unspec}_{\Psi,\,K} & = & \mathsf{unspec}_\Psi,\, ? \end{array}$$

Definition D.7 *The partial context specified solely at i is defined as follows.*

$\boxed{\Psi \widehat{@} i}$

$$\begin{array}{lcl} (\Psi,\, K)\,\widehat{@}\, i & = & \mathsf{unspec}_\Psi,\, K \text{ when } |\Psi| = i \\ (\Psi,\, K)\,\widehat{@}\, i & = & (\Psi \widehat{@} i),\, ? \text{ when } |\Psi| > i \end{array}$$

Definition D.8 *Joining two partial contexts is defined as follows.*

$\boxed{\widehat{\Psi} \circ \widehat{\Psi}'}$

$$\begin{array}{lcl} \bullet \circ \bullet & = & \bullet \\ (\widehat{\Psi},\, K) \circ (\widehat{\Psi}',\, K) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\, K \\ (\widehat{\Psi},\, ?) \circ (\widehat{\Psi}',\, K) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\, K \\ (\widehat{\Psi},\, K) \circ (\widehat{\Psi}',\, ?) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\, K \\ (\widehat{\Psi},\, ?) \circ (\widehat{\Psi}',\, ?) & = & (\widehat{\Psi} \circ \widehat{\Psi}'),\, ? \end{array}$$

Definition D.9 *The notion of one partial context being a less precise version of another one is defined as follows.*

$\boxed{\widehat{\Psi} \sqsubseteq \widehat{\Psi}'}$

$$\begin{array}{l} \bullet \sqsubseteq \bullet \\ (\widehat{\Psi},\, K) \sqsubseteq (\widehat{\Psi}',\, K) \quad \Leftarrow \quad \widehat{\Psi} \sqsubseteq \widehat{\Psi}' \\ (\widehat{\Psi},\, ?) \sqsubseteq (\widehat{\Psi}',\, K) \quad \Leftarrow \quad \widehat{\Psi} \sqsubseteq \widehat{\Psi}' \\ (\widehat{\Psi},\, ?) \sqsubseteq (\widehat{\Psi}',\, ?) \quad \Leftarrow \quad \widehat{\Psi} \sqsubseteq \widehat{\Psi}' \end{array}$$

Definition D.10 *We define a judgement to extract the relevant extension variables out of a context.*

$$\boxed{\mathsf{relevant}\,(\Psi \vdash T : K) = \widehat{\Psi}}$$

$$\mathsf{relevant}\left(\frac{\Psi;\ \Phi \vdash t : t' \qquad \Psi;\ \Phi \vdash t' : s}{\Psi \vdash [\Phi]t : [\Phi]t'}\right) = \mathsf{relevant}\left(\Psi;\ \Phi \vdash t : t'\right)$$

$$\mathsf{relevant}\left(\frac{\Psi \vdash \Phi,\ \Phi'\ \mathrm{wf}}{\Psi \vdash [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}}\right) = \mathsf{relevant}\left(\Psi \vdash \Phi,\ \Phi'\ \mathrm{wf}\right)$$

$$\boxed{\mathsf{relevant}\,(\Psi \vdash \Phi\ \mathrm{wf}) = \widehat{\Psi}}$$

$$\mathsf{relevant}\left(\frac{}{\Psi \vdash \bullet\ \mathrm{wf}}\right) = \mathsf{unspec}_{\Psi} \qquad \mathsf{relevant}\left(\frac{\Psi \vdash \Phi\ \mathrm{wf} \qquad \Psi;\ \Phi \vdash t : s}{\Psi \vdash (\Phi,\ t)\ \mathrm{wf}}\right) = \mathsf{relevant}\,(\Psi;\ \Phi \vdash t : s)$$

$$\mathsf{relevant}\left(\frac{\Psi \vdash \Phi\ \mathrm{wf} \qquad (\Psi).i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash (\Phi,\ X_i)\ \mathrm{wf}}\right) = \mathsf{relevant}\,(\Psi \vdash \Phi\ \mathrm{wf}) \circ (\Psi \widehat{@}\, i)$$

$$\boxed{\text{relevant}\,(\Psi;\ \Phi \vdash t : t') = \widehat{\Psi}}$$

$$\text{relevant}\left(\frac{c : t \in \Sigma}{\Psi;\ \Phi \vdash c : t}\right) = \text{relevant}\,(\Psi \vdash \Phi\ \text{wf}) \qquad \text{relevant}\left(\frac{\Phi.\mathbf{I} = t}{\Psi;\ \Phi \vdash f_{\mathbf{I}} : t}\right) = \text{relevant}\,(\Psi \vdash \Phi\ \text{wf})$$

$$\text{relevant}\left(\frac{(s,s') \in \mathcal{A}}{\Psi;\ \Phi \vdash s : s'}\right) = \text{relevant}\,(\Psi \vdash \Phi\ \text{wf})$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s,s',s'') \in \mathcal{R}}{\Psi;\ \Phi \vdash \Pi(t_1).t_2 : s''}\right) = \text{relevant}\left(\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : s'\right)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi;\ \Phi \vdash \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|,\,.} : s'}{\Psi;\ \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\ \lfloor t' \rfloor_{|\Phi|,\,.}}\right) =$$

$$\text{relevant}\left(\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil_{|\Phi|} : t'\right)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_1 : \Pi(t).t' \qquad \Psi;\ \Phi \vdash t_2 : t}{\Psi;\ \Phi \vdash t_1\ t_2 : \left\lceil t' \right\rceil_{|\Phi|} \cdot (\text{id}_\Phi, t_2)}\right) = \text{relevant}\left(\Psi;\ \Phi \vdash t_1 : \Pi(t).t'\right) \circ \text{relevant}\,(\Psi;\ \Phi \vdash t_2 : t)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_1 : t \qquad \Psi;\ \Phi \vdash t_2 : t \qquad \Psi;\ \Phi \vdash t : \text{Type}}{\Psi;\ \Phi \vdash t_1 = t_2 : \text{Prop}}\right) =$$

$$\text{relevant}\,(\Psi;\ \Phi \vdash t_1 : t) \circ \text{relevant}\,(\Psi;\ \Phi \vdash t_2 : t) \qquad \text{relevant}\left(\frac{(\Psi).i = T \qquad T = [\Phi']t' \qquad \Psi;\ \Phi \vdash \sigma : \Phi'}{\Psi;\ \Phi \vdash X_i/\sigma : t' \cdot \sigma}\right) =$$

$$(\text{relevant}\left(\Psi\lfloor_i \vdash [\Phi']t' : [\Phi']s\right), \overbrace{?, ?, \cdots, ?}^{|\Psi| - i \text{ times}}) \circ \text{relevant}\left(\Psi;\ \Phi \vdash \sigma : \Phi'\right) \circ (\Psi \widehat{@} i)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t : t_1 \qquad \Psi;\ \Phi \vdash t_1 : \text{Prop} \qquad \Psi;\ \Phi \vdash t' : t_1 = t_2}{\Psi;\ \Phi \vdash \text{conv}\ t\ t' : t_2}\right) =$$

$$\text{relevant}\,(\Psi;\ \Phi \vdash t : t_1) \circ \text{relevant}\left(\Psi;\ \Phi \vdash t' : t_1 = t_2\right)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_1 : t \qquad \Psi;\ \Phi \vdash t_1 = t_1 : \text{Prop}}{\Psi;\ \Phi \vdash \text{refl}\ t_1 : t_1 = t_1}\right) = \text{relevant}\,(\Psi;\ \Phi \vdash t_1 : t)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_a : t_1 = t_2}{\Psi;\ \Phi \vdash \text{symm}\ t_a : t_2 = t_1}\right) = \text{relevant}\,(\Psi;\ \Phi \vdash t_a : t_1 = t_2)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_a : t_1 = t_2 \qquad \Psi;\ \Phi \vdash t_b : t_2 = t_3}{\Psi;\ \Phi \vdash \text{trans}\ t_a\ t_b : t_1 = t_3}\right) =$$

$$\text{relevant}\,(\Psi;\ \Phi \vdash t_a : t_1 = t_2) \circ \text{relevant}\,(\Psi;\ \Phi \vdash t_b : t_2 = t_3)$$

$$\text{relevant}\left(\frac{\Psi;\ \Phi \vdash t_a : M_1 = M_2 \qquad \Psi;\ \Phi \vdash M_1 : A \to B \qquad \Psi;\ \Phi \vdash t_b : N_1 = N_2 \qquad \Psi;\ \Phi \vdash N_1 : A}{\Psi;\ \Phi \vdash \text{congapp}\ t_a\ t_b : M_1\ N_1 = M_2\ N_2}\right) =$$

$$\text{relevant}\,(\Psi;\ \Phi \vdash t_a : M_1 = M_2) \circ \text{relevant}\,(\Psi;\ \Phi \vdash t_b : N_1 = N_2)$$

$$\mathsf{relevant}\left(\frac{\Psi;\,\Phi\vdash t_a : A_1 = A_2 \qquad \Psi;\,\Phi, A_1 \vdash \lceil t_b\rceil : B_1 = B_2 \qquad \Psi;\,\Phi\vdash A_1 : \mathsf{Prop} \qquad \Psi;\,\Phi, A_1 \vdash \lceil B_1\rceil : \mathsf{Prop}}{\Psi;\,\Phi\vdash \mathsf{congimpl}\; t_a\; (\lambda(A_1).t_b) : \Pi(A_1).\lfloor B_1\rfloor = \Pi(A_2).\lfloor B_2\rfloor}\right) =$$

$$\mathsf{relevant}\,(\Psi;\,\Phi\vdash t_a : A_1 = A_2) \circ \mathsf{relevant}\,(\Psi;\,\Phi, A_1 \vdash \lceil t_b\rceil : B_1 = B_2)$$

$$\mathsf{relevant}\left(\frac{\Psi;\,\Phi,\, A\vdash \lceil t_b\rceil : B = B' \qquad \Psi;\,\Phi\vdash \Pi(A).\lfloor B\rfloor = \Pi(A).\lfloor B'\rfloor : \mathsf{Prop}}{\Psi;\,\Phi\vdash \mathsf{congpi}\;(\lambda(A).t_b) : \Pi(A).\lfloor B\rfloor = \Pi(A).\lfloor B'\rfloor}\right) =$$

$$\mathsf{relevant}\left(\Psi;\,\Phi,\, A\vdash \lceil t_b\rceil : B = B'\right)$$

$$\mathsf{relevant}\left(\frac{\Psi;\,\Phi,\, A\vdash \lceil t_b\rceil : B_1 = B_2 \qquad \Psi;\,\Phi\vdash \lambda(A).\lfloor B_1\rfloor = \lambda(A).\lfloor B_2\rfloor : \mathsf{Prop}}{\Psi;\,\Phi\vdash \mathsf{conglam}\;(\lambda(A).t_b) : \lambda(A).\lfloor B_1\rfloor = \lambda(A).\lfloor B_2\rfloor}\right) =$$

$$\mathsf{relevant}\,(\Psi;\,\Phi,\, A\vdash \lceil t_b\rceil : B_1 = B_2)$$

$$\mathsf{relevant}\left(\frac{\Psi;\,\Phi\vdash \lambda(A).M : A\to B \qquad \Psi;\,\Phi\vdash N : A \qquad \Psi;\,\Phi\vdash A\to B : \mathsf{Type}}{\Psi;\,\Phi\vdash \mathsf{beta}\;(\lambda(A).M)\;N : (\lambda(A).M)\;N = \lceil M\rceil\cdot(\mathsf{id}_\Phi, N)}\right) =$$

$$\mathsf{relevant}\,(\Psi;\,\Phi\vdash \lambda(A).M : A\to B) \circ \mathsf{relevant}\,(\Psi;\,\Phi\vdash N : A)$$

$\boxed{\Psi;\,\Phi\vdash\sigma : \Phi'}$

$$\mathsf{relevant}\left(\frac{}{\Psi;\,\Phi\vdash\bullet:\bullet}\right) = \mathsf{relevant}\,(\Psi\vdash\Phi\;\mathrm{wf})$$

$$\mathsf{relevant}\left(\frac{\Psi;\,\Phi\vdash\sigma:\Phi' \qquad \Psi;\,\Phi\vdash t : t'\cdot\sigma}{\Psi;\,\Phi\vdash(\sigma,\,t):(\Phi',\,t')}\right) = \mathsf{relevant}\left(\Psi;\,\Phi\vdash\sigma:\Phi'\right)\circ\mathsf{relevant}\left(\Psi;\,\Phi\vdash t : t'\cdot\sigma\right)$$

$$\mathsf{relevant}\left(\frac{\Psi;\,\Phi\vdash\sigma:\Phi' \qquad (\Psi).i = [\Phi']\,\mathsf{ctx} \qquad \Phi',\, X_i\subseteq\Phi}{\Psi;\,\Phi\vdash(\sigma,\,\mathbf{id}(X_i)):(\Phi',\,X_i)}\right) = \mathsf{relevant}\left(\Psi;\,\Phi\vdash\sigma:\Phi'\right)$$

Lemma D.11 (More-informed contexts preserve judgements) *Assuming $\widehat{\Psi}\sqsubseteq\widehat{\Psi}'$:*

1. *If $\widehat{\Psi}\vdash T : K$ then $\widehat{\Psi}'\vdash T : K$.*
2. *If $\widehat{\Psi}\vdash\Phi$ wf then $\widehat{\Psi}'\vdash\Phi$ wf.*
3. *If $\widehat{\Psi};\,\Phi\vdash t : t'$ then $\widehat{\Psi}';\,\Phi\vdash t : t'$.*
4. *If $\widehat{\Psi};\,\Phi\vdash\sigma:\Phi'$ then $\widehat{\Psi}';\,\Phi\vdash\sigma:\Phi'$.*

Simple by structural induction on the judgements. The interesting cases are the ones mentioning extension variables, as for example when $\Phi = \Phi', X_i$, or $t = X_i/\sigma$. In both such cases, the typing rule has a side condition requiring that $\widehat{\Psi}.i = T$. Since $\widehat{\Psi}\sqsubseteq\widehat{\Psi}'$, we have that $\widehat{\Psi}'.i = T$.

Lemma D.12 (Relevancy is decidable) *1. If $\Psi\vdash T : K$, then there exists a unique $\widehat{\Psi}$ such that relevant$(\Psi\vdash T : K) = \widehat{\Psi}$.*

2. *If $\Psi\vdash\Phi$ wf, then there exists a unique $\widehat{\Psi}$ such that relevant$(\Psi\vdash\Phi\;wf) = \widehat{\Psi}$.*
3. *If $\Psi;\,\Phi\vdash t : t'$, then there exists a unique $\widehat{\Psi}$ such that relevant$(\Psi;\,\Phi\vdash t : t') = \widehat{\Psi}$.*

4. If $\Psi;\ \Phi \vdash \sigma : \Phi'$*, then there exists a unique* $\widehat{\Psi}$ *such that* $\mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi') = \widehat{\Psi}$.

The relevancy judgements are defined by structural induction on the corresponding typing derivations. It is crucial to take into account the fact that $\vdash \Psi$ wf and $\Psi \vdash \Phi$ wf are implicitly present along any typing derivation that mentions such contexts; thus these derivations themselves, as well as their sub-derivations, are structurally included in derivations like $\Psi;\ \Phi \vdash t : t'$. Furthermore, it is easy to see that all the joins used are defined, since in most cases two results of the relevancy procedure on a judgement using the same context Ψ are joined, which is always well-defined. The only case where this does not hold (use of extension variables in terms), the joins are still defined because of the adaptation of the resulting $\widehat{\Psi}$ by affixing the unspecified elements.

Lemma D.13 (Properties of context join) *1.* $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 \sqsubseteq \widehat{\Psi}_1$

2. $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 \sqsubseteq \widehat{\Psi}_2$

3. $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 = \widehat{\Psi}_2 \circ \widehat{\Psi}_1$

4. $(\widehat{\Psi}_1 \circ \widehat{\Psi}_2) \circ \widehat{\Psi}_3 = \widehat{\Psi}_1 \circ (\widehat{\Psi}_2 \circ \widehat{\Psi}_3)$

5. If $\widehat{\Psi}_1 \sqsubseteq \widehat{\Psi}_2$ *then* $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 = \widehat{\Psi}_2$

6. If $\widehat{\Psi}_1 \sqsubseteq \widehat{\Psi}'_1$ *then* $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 \sqsubseteq \widehat{\Psi}'_1 \circ \widehat{\Psi}_2$

All are simple to prove by induction.

Lemma D.14 (Relevancy when weakening the extensions context) *1. If* $\Psi \vdash T : K$*, then* $\mathsf{relevant}(\Psi,\ \Psi' \vdash T : K) = \mathsf{relevant}(\Psi \vdash T : K), \overbrace{?, \cdots, ?}^{|\Psi'|}$.

2. If $\Psi \vdash \Phi$ *wf, then* $\mathsf{relevant}(\Psi,\ \Psi' \vdash \Phi\ \mathit{wf}) = \mathsf{relevant}(\Psi \vdash \Phi\ \mathit{wf}), \overbrace{?, \cdots, ?}^{|\Psi'|}$.

3. If $\Psi;\ \Phi \vdash t : t'$*, then* $\mathsf{relevant}(\Psi,\ \Psi';\ \Phi \vdash t : t') = \mathsf{relevant}(\Psi;\ \Phi \vdash t : t'), \overbrace{?, \cdots, ?}^{|\Psi'|}$.

4. If $\Psi;\ \Phi \vdash \sigma : \Phi'$*, then* $\mathsf{relevant}(\Psi,\ \Psi';\ \Phi \vdash \sigma : \Phi') = \mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi'), \overbrace{?, \cdots, ?}^{|\Psi'|}$.

Simple to prove by induction.

Lemma D.15 (Relevancy of sub-judgements is implied) *1.(a)* $\mathsf{relevant}(\Psi \vdash \Phi\ \mathit{wf}) \sqsubseteq \mathsf{relevant}(\Psi \vdash \Phi,\ \Phi'\ \mathit{wf})$

(b) $\mathsf{relevant}(\Psi \vdash \Phi\ \mathit{wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t : t')$

(c) $\mathsf{relevant}(\Psi \vdash \Phi\ \mathit{wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi')$.

2.(a) If $\Psi;\ \Phi \vdash t : t'$ *then* $\mathsf{relevant}(\Psi;\ \Phi,\ \Phi' \vdash t : t') = \mathsf{relevant}(\Psi;\ \Phi \vdash t : t') \circ \mathsf{relevant}(\Psi \vdash \Phi,\ \Phi'\ \mathit{wf})$.

(b) If $\Psi;\ \Phi \vdash \sigma : \Phi'$ *then* $\mathsf{relevant}(\Psi;\ \Phi,\ \Phi'' \vdash \sigma : \Phi') = \mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi') \circ \mathsf{relevant}(\Psi \vdash \Phi,\ \Phi''\ \mathit{wf})$.

3.(a) If $\Psi;\ \Phi \vdash t : t'$ *and* $\Psi;\ \Phi \vdash t' : s$ *then* $\mathsf{relevant}(\Psi;\ \Phi \vdash t' : s) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t : t')$.

(b) If $\Psi;\ \Phi' \vdash \sigma : \Phi$ *then* $\mathsf{relevant}(\Psi \vdash \Phi\ \mathit{wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$.

4.(a) If $\Psi;\ \Phi \vdash t : t'$ *and* $\Psi;\ \Phi' \vdash \sigma : \Phi$*, then* $\mathsf{relevant}(\Psi;\ \Phi' \vdash t \cdot \sigma : t' \cdot \sigma) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t : t') \circ \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$.

(b) If $\Psi;\ \Phi' \vdash \sigma : \Phi$ *and* $\Psi;\ \Phi'' \vdash \sigma' : \Phi'$*, then* $\mathsf{relevant}(\Psi;\ \Phi'' \vdash \sigma \cdot \sigma' : \Phi) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi) \circ \mathsf{relevant}(\Psi;\ \Phi'' \vdash \sigma' : \Phi')$.

Part 1(a) Trivial by induction the derivation of relevancy.

Part 1(b) By inversion of the derivation of $\mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf}) = \widehat{\Psi}$. In the base cases, this is directly proved by the relevancy judgement; in the case where we have $\mathsf{relevant}(\Psi;\ \Phi \vdash t : t') = \mathsf{relevant}(\Psi,\ \Phi,\ t_1 \vdash t_2 : t_3)$, by induction hypothesis get that $\mathsf{relevant}(\Psi \vdash \Phi,\ t_1\ \mathrm{wf})$, which by inversion gives us the desired; in the metavariables case trivially follows from repeated inversions of $\mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi')$.

Part 1(c) Trivial by induction and use of part 1(b).

Part 2 By induction on the typing derivations of t and t' all cases follow trivially.

Part 3(a) By induction on the derivation of $\Psi;\ \Phi \vdash t : t'$.

Case $t = c\ \triangleright$ Simply using the above parts and the fact that $\Psi;\ \bullet \vdash t' : s$, we have that $\mathsf{relevant}(\Psi;\ \Phi \vdash t' : s) = \mathsf{unspec}_\Psi \circ \mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf}) = \mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t : t')$.

Case $t = s\ \triangleright$ Similarly as the above case.

Case $t = v_{\mathbf{I}}\ \triangleright$ We have that $\Psi;\ \Phi|_{\mathbf{I}} \vdash \Phi.\mathbf{I} : s$, by inversion of the well-formedness derivation for Φ. Therefore $\mathsf{relevant}(\Psi;\ \Phi \vdash t' : s) = \mathsf{relevant}(\Psi;\ \Phi|_{\mathbf{I}} \vdash t' : s) \circ \mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf})$. By repeated inversion of $\Psi \vdash \Phi$ wf we get that $\mathsf{relevant}(\Psi \vdash (\Phi|_{\mathbf{I}},\ \Phi.\mathbf{I})\ \mathrm{wf}) \sqsubseteq \mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf})$. Thus we have that $\mathsf{relevant}(\Psi;\ \Phi \vdash t' : s) \sqsubseteq \mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf})$, which proves the desired.

Case $t = \Pi(t_1).t_2\ \triangleright$ Trivially from the fact that $\mathsf{relevant}(\Psi;\ \Phi \vdash s'' : s''') = \mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi, t_1 \vdash \lceil t_2 \rceil)$

Case $t = \lambda(t_1).t_2\ \triangleright$ We have that $\mathsf{relevant}(\Psi;\ \Phi \vdash \Pi(t_1).\lfloor t' \rfloor : s') = \mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash \lceil \lfloor t' \rfloor \rceil : s) = \mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash t' : s)$. So by induction hypothesis, since $\Psi;\ \Phi,\ t_1 \vdash t' : s$ is a sub-derivation in $\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil : t'$, we have that $\mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash t' : s) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash t_2 : t')$, which is the desired.

Case $t = t_1\ t_2\ \triangleright$ By induction hypothesis get that $\mathsf{relevant}(\Psi;\ \Phi \vdash \Pi(t).t' : s) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t_1 : \Pi(t).t')$. (*Here we assume unique typing for* Π *types*). Furthermore, we have that $\mathsf{relevant}(\Psi;\ \Phi \vdash \Pi(t).t' : s) = \mathsf{relevant}(\Psi;\ \Phi,\ t \vdash \lceil t' \rceil : s)$. Otherwise, it is simple to prove that $\mathsf{relevant}(\Psi;\ \Phi \vdash (\mathsf{id}_\Phi,\ t_2) : (\Phi,\ \lceil t' \rceil)) = \mathsf{relevant}(\Psi;\ \Phi \vdash t_2 : \lceil t' \rceil)$, thus the desired follows trivially following part 4.

Case $t = X_i/\sigma\ \triangleright$ We have that $\mathsf{relevant}(\Psi;\ \Phi' \vdash t' : s) = \mathsf{relevant}(\Psi\lfloor_i;\ \Phi' \vdash t' : s),\ \overbrace{?, \cdots, ?}^{|\Psi| - i \text{ times}}$ from inversion of well-formedness for Ψ. Furthermore, we have that $\Psi;\ \Phi \vdash \sigma : \Phi'$ from typing inversion. Thus, using part 4, we get that $\mathsf{relevant}(\Psi;\ \Phi \vdash t' \cdot \sigma : s) \sqsubseteq (\mathsf{relevant}(\Psi\lfloor_i;\ \Phi' \vdash t' : s),\ \overbrace{?, \cdots, ?}^{|\Psi| - i \text{ times}}) \circ \mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi')$. Thus the result follows directly, taking the properties of join into account.

Case $t = (t_1 = t_2)\ \triangleright$ Trivial.

Case $t = \mathsf{conv}\ t\ t'\ \triangleright$ By induction hypothesis we get that $\mathsf{relevant}(\Psi;\ \Phi \vdash t_1 = t_2 : \mathsf{Prop}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t' : t_1 = t_2)$. By inversion of relevancy for $t_1 = t_2$ we get that it is equal to $\mathsf{relevant}(\Psi;\ \Phi \vdash t_1 : \mathsf{Prop}) \circ \mathsf{relevant}(\Psi;\ \Phi \vdash t_2 : \mathsf{Prop}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t_2 : \mathsf{Prop})$. Thus the desired follows trivially using the properties of joining contexts.

Case (rest) $\triangleright$ Following the techniques used above.

Part 3(b) By induction on the derivation of $\Psi;\ \Phi' \vdash \sigma : \Phi$.

Case $\sigma = \bullet$ $\triangleright$ Trivial.

Case $\sigma = \sigma', t'$ $\triangleright$ By induction hypothesis for σ', use of part 3(a) for t', and definition of relevancy for Φ.

Case $\sigma = \sigma', \mathbf{id}(X_i)$ $\triangleright$ By induction hypothesis for σ', and also using the side condition for X_i being part of Φ': by inversion of well-formedness for Φ', we get that $\Psi\widehat{@}i \sqsubseteq \mathsf{relevant}(\Psi \vdash \Phi' \text{ wf})$ and thus also $\Psi\widehat{@}i \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$, proving the desired.

Part 4(a) By induction on the typing derivation for t.

Case $t = c$ $\triangleright$ We have that $\mathsf{relevant}(\Psi;\ \Phi \vdash c : t') = \mathsf{relevant}(\Psi \vdash \Phi \text{ wf})$, and $\mathsf{relevant}(\Psi;\ \Phi' \vdash c \cdot \sigma : t' \cdot \sigma) = \mathsf{relevant}(\Psi \vdash \Phi' \text{ wf})$. We need to show that $\mathsf{relevant}(\Psi \vdash \Phi \text{ wf}) \sqsubseteq \mathsf{relevant}(\Psi \vdash \Phi' \text{ wf}) \circ \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$. We have that $\mathsf{relevant}(\Psi \vdash \Phi' \text{ wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$, so the join in the above equality is well-defined; from the properties of join it is evident that it is enough to show $\mathsf{relevant}(\Psi \vdash \Phi \text{ wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$. This is trivially proved by part 3(b).

Case $t = s$ $\triangleright$ Similarly.

Case $t = f_{\mathbf{I}}$ $\triangleright$ We have that $\mathsf{relevant}(\Psi;\ \Phi' \vdash f_{\mathbf{I}} \cdot \sigma : t' \cdot \sigma) = \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma.i : t' \cdot \sigma)$. By inversion for σ, we have that $\mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma.i : \Phi.i \cdot \sigma) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$. Thus the desired directly follows.

Case $t = \Pi(t_1).t_2$ $\triangleright$ By induction hypothesis for $\lceil t_2 \rceil$ and $\sigma = \sigma,\ f_{|\Phi|}$, we get that:
$\mathsf{relevant}\big(\Psi;\ \Phi', t_1 \cdot \sigma \vdash \lceil t_2 \rceil \cdot (\sigma,\ f_{|\Phi|}) : s''\big) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi',\ t_1 \cdot \sigma \vdash \lceil t_2 \rceil : s'') \circ \mathsf{relevant}\big(\Psi;\ \Phi',\ t_1 \cdot \sigma \vdash (\sigma,\ f_{|\Phi|}) : (\Phi,\ t_1)\big)$.
Also we have that $\mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi',\ t_1 \cdot \sigma \vdash \sigma : \Phi)$. Using the known properties of freshening and substitutions, we know that $\mathsf{relevant}(\Psi;\ \Phi' \vdash t \cdot \sigma : s'') = \mathsf{relevant}\big(\Psi;\ \Phi', t_1 \cdot \sigma \vdash \lceil t_2 \rceil \cdot (\sigma,\ f_{|\Phi|}) : s''\big)$, thus this is the desired.

Case $t = \lambda(t_1).t_2$ $\triangleright$ Similar to the above.

Case $t = t_1\ t_2$ $\triangleright$ By induction hypothesis we get that:
$\mathsf{relevant}(\Psi;\ \Phi' \vdash t_1 \cdot \sigma : \Pi(t \cdot \sigma).(t' \cdot \sigma)) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t_1 : \Pi(t).t') \circ \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$, and that
$\mathsf{relevant}(\Psi;\ \Phi' \vdash t_2 \cdot \sigma : t \cdot \sigma) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi \vdash t_2 : t) \circ \mathsf{relevant}(\Psi;\ \Phi' \vdash \sigma : \Phi)$. Furthermore, we have that
$\mathsf{relevant}(\Psi;\ \Phi' \vdash t_1 \cdot \sigma\ t_2 \cdot \sigma : \lceil t' \cdot \sigma \rceil \cdot (\mathsf{id}_{\Phi'},\ t_2 \cdot \sigma))$
$= \mathsf{relevant}(\Psi;\ \Phi' \vdash t_1 \cdot \sigma : \Pi(t \cdot \sigma).(t' \cdot \sigma)) \circ \mathsf{relevant}(\Psi;\ \Phi' \vdash t_2 \cdot \sigma : t \cdot \sigma)$. The desired follows trivially, using the properties of join.

Case $t = X_i/\sigma'$ $\triangleright$ Trivial, using part 4(b).

Case (rest) $\triangleright$ Using similar techniques as above.

Part 4(b) By induction and use of part 4(a).

Lemma D.16 (Relevancy soundness) *1. If* $\Psi \vdash T : K$ *and* $\mathsf{relevant}(\Psi \vdash T : K) = \widehat{\Psi}$ *then* $\widehat{\Psi} \vdash T : K$.

2. If $\Psi \vdash \Phi$ *wf and* $\mathsf{relevant}(\Psi \vdash \Phi \text{ wf}) = \widehat{\Psi}$ *then* $\widehat{\Psi} \vdash \Phi$ *wf.*

3. If $\Psi;\ \Phi \vdash t : t'$ *and* $\mathsf{relevant}(\Psi;\ \Phi \vdash t : t') = \widehat{\Psi}$ *then* $\widehat{\Psi};\ \Phi \vdash t : t'$.

4. If $\Psi;\ \Phi \vdash \sigma : \Phi'$ *and* $\mathsf{relevant}(\Psi;\ \Phi \vdash \sigma : \Phi') = \widehat{\Psi}$ *then* $\widehat{\Psi};\ \Phi \vdash \sigma : \Phi'$.

Part 1 By induction on the derivation of $\Psi \vdash T : K$.

Case $T = [\Phi]t \ \triangleright$ By part 3 we have that if $\mathsf{relevant}(\Psi;\ \Phi \vdash t : t') = \widehat{\Psi}$, then $\widehat{\Psi};\ \Phi \vdash t : t'$. From this we also get that $\widehat{\Psi};\ \Phi \vdash t' : s$, and thus it is trivial to construct a derivation of $\widehat{\Psi} \vdash [\Phi]t : [\Phi]t'$.

Case $T = [\Phi]\Phi' \ \triangleright$ From part 2 we get that if $\mathsf{relevant}(\Psi \vdash \Phi,\ \Phi'\ \mathrm{wf}) = \widehat{\Psi}$, then $\widehat{\Psi} \vdash \Phi,\ \Phi'$ wf, thus the desired follows trivially.

Part 2 By induction on the derivation of $\Psi \vdash \Phi$ wf.

Case $\Phi = \bullet \ \triangleright$ Trivially we have that $\mathsf{unspec}_\Psi \vdash \bullet$ wf.

Case $\Phi = \Phi,\ t \ \triangleright$ We have that if $\mathsf{relevant}(\Psi;\ \Phi \vdash t : s) = \widehat{\Psi}$, then $\widehat{\Psi};\ \Phi \vdash t : s$ by part 3, and furthermore using the implicit requirement that Φ is well-formed, we also get that $\widehat{\Psi} \vdash \Phi$ wf. Thus using the appropriate typing rule we get $\widehat{\Psi} \vdash (\Phi,\ t)$ wf.

Case $\Phi = \Phi,\ X_i \ \triangleright$ By induction we get that if $\mathsf{relevant}(\Psi \vdash \Phi\ \mathrm{wf}) = \widehat{\Psi}$, then $\widehat{\Psi} \vdash \Phi$ wf, and thus also $\widehat{\Psi} \circ (\Psi \widehat{@} i) \vdash \Phi$ wf. Furthermore, $(\widehat{\Psi} \circ (\Psi \widehat{@} i)).i = \Psi.i$. Thus using the appropriate well-formedness rule for Φ we get that $\widehat{\Psi} \vdash (\Phi,\ X_i)$ wf.

Part 3 By induction on the derivation of $\Psi;\ \Phi \vdash t : t'$.

Case $t = c \ \triangleright$ Trivially we have that $\widehat{\Psi};\ \Phi \vdash c : t$ for any $\widehat{\Psi}$, Φ such that $\widehat{\Psi} \vdash \Phi$ wf, which holds for the corresponding $\widehat{\Psi}$ based on part 2.

Case $t = s \ \triangleright$ Similarly as above.

Case $t = f_{\mathbf{I}} \ \triangleright$ Again, as above.

Case $t = \Pi(t_1).t_2 \ \triangleright$ Simple by induction hypothesis for $\lceil t_2 \rceil$, and also from the fact that $\mathsf{relevant}(\Psi;\ \Phi \vdash t_1 : s) \sqsubseteq \mathsf{relevant}(\Psi \vdash (\Phi,\ t_1)\ \mathrm{wf}) \sqsubseteq \mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil : s')$.

Case $t = \lambda(t_1).t_2 \ \triangleright$ By induction hypothesis for $\lceil t_2 \rceil$, if $\mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash \lceil t_2 \rceil : s') = \widehat{\Psi}$, we get that $\widehat{\Psi};\ \Phi,\ t_1 \vdash \lceil t_2 \rceil : s'$. Thus we also have that $\widehat{\Psi};\ \Phi \vdash t_1 : s$, and also that either $t' = \mathsf{Type}'$ (which is an impossible case), or $\widehat{\Psi};\ \Phi,\ t_1 \vdash t' : s''$. By inversion of typing for $\Psi;\ \Phi \vdash \Pi(t_1).\lfloor t' \rfloor : s'$ we get that in fact $s'' = s'$, and thus it is easy to derive $\widehat{\Psi};\ \Phi,\ t_1 \vdash t' : s'$ and $\widehat{\Psi};\ \Phi \vdash \Pi(t_1).\lfloor t' \rfloor : s'$. From these we get the desired derivation for $\widehat{\Psi};\ \Phi \vdash \lambda(t_1).t_2 : \Pi(t_1).\lfloor t' \rfloor$.

Case $t = t_1\ t_2 \ \triangleright$ Trivial by induction hypothesis for t_1 and t_2.

Case $t = (t_1 = t_2) \ \triangleright$ Again, trivial by induction hypothesis for t_1 and t_2, and also from the fact that $\widehat{\Psi}_1;\ \Phi \vdash t_1 : t$ implies $\widehat{\Psi}_1;\ \Phi \vdash t : \mathsf{Type}$.

Case $t = X_i/\sigma \ \triangleright$ From the first part (relevancy of T under the prefix context), we get that $\vdash \widehat{\Psi}$ wf. Furthermore, using part 4 we get that $\widehat{\Psi};\ \Phi \vdash \sigma : \Phi'$. Last, it is trivial to derive $\widehat{\Psi};\ \Phi \vdash X_i/\sigma : t' \cdot \sigma$ using the same typing rule, since $\widehat{\Psi}.i = \Psi.i$.

Part 4 By induction on the derivation of $\Psi;\ \Phi \vdash \sigma : \Phi'$.

Case $\sigma = \bullet$ $\triangleright$ Trivial.

Case $\sigma = \sigma', t$ $\triangleright$ Trivial by induction hypothesis and use of part 3.

Case $\sigma = \sigma', \mathbf{id}(X_i)$ $\triangleright$ By induction hypothesis get $\widehat{\Psi};\ \Phi \vdash \sigma : \Phi'$. Furthermore, from $\widehat{\Psi} \vdash \Phi$ wf and the fact that $\Phi',\ X_i \subseteq \Phi$, we have by repeated typing inversions that $\Psi @ i \sqsubseteq \widehat{\Psi}$. Thus $\widehat{\Psi}.i = \Psi.i$, and we can construct a derivation for $\widehat{\Psi};\ \Phi \vdash (\sigma,\ \mathbf{id}(X_i)) : (\Phi',\ X_i)$.

Definition D.17 *Applying an extension substitution to a partial context is defined as follows, assuming that the partial context does not contain extension variables bigger than* $X_{|\Psi|-1}$.

$\boxed{\widehat{\Psi} \cdot \sigma_\Psi}$

$$
\begin{array}{lcl}
\bullet \cdot \sigma_\Psi & = & \bullet \\
(\widehat{\Psi},\ K) \cdot \sigma_\Psi & = & \widehat{\Psi} \cdot \sigma_\Psi,\ K \cdot (\sigma_\Psi,\ X_{|\Psi|},\ \cdots,\ X_{|\Psi|+|\widehat{\Psi}|}) \\
(\widehat{\Psi},\ ?) \cdot \sigma_\Psi & = & \widehat{\Psi} \cdot \sigma_\Psi,\ ?
\end{array}
$$

Lemma D.18 (Relevancy and extension substitution) *1. If* $\mathsf{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathsf{relevant}(\Psi,\ \Psi_u \vdash T : K)$, $\Psi' \vdash \sigma_\Psi : \Psi$, *and* $\sigma'_\Psi = \sigma_\Psi,\ X_{|\Psi'|},\ \cdots,\ X_{|\Psi'|+|\Psi_u|}$, *then* $\mathsf{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash T \cdot \sigma'_\Psi : K \cdot \sigma'_\Psi)$.

2. If $\mathsf{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathsf{relevant}(\Psi,\ \Psi_u \vdash \Phi\ wf)$, $\Psi' \vdash \sigma_\Psi : \Psi$, *and* $\sigma'_\Psi = \sigma_\Psi,\ X_{|\Psi'|},\ \cdots,\ X_{|\Psi'|+|\Psi_u|}$, *then* $\mathsf{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash \Phi \cdot \sigma'_\Psi\ wf)$.

3. If $\mathsf{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathsf{relevant}(\Psi,\ \Psi_u;\ \Phi \vdash t : t')$, $\Psi' \vdash \sigma_\Psi : \Psi$, *and* $\sigma'_\Psi = \sigma_\Psi,\ X_{|\Psi'|},\ \cdots,\ X_{|\Psi'|+|\Psi_u|}$, *then* $\mathsf{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}(\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash t \cdot \sigma'_\Psi : t' \cdot \sigma'_\Psi)$.

4. If $\mathsf{unspec}_\Psi, \widehat{\Psi}_u \sqsubseteq \mathsf{relevant}(\Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi')$, $\Psi' \vdash \sigma_\Psi : \Psi$, *and* $\sigma'_\Psi = \sigma_\Psi,\ X_{|\Psi'|},\ \cdots,\ X_{|\Psi'|+|\Psi_u|}$, *then* $\mathsf{unspec}_{\Psi'}, \widehat{\Psi}_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}(\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash \sigma \cdot \sigma'_\Psi : \Phi' \cdot \sigma'_\Psi)$.

Part 1 By induction on the typing derivation of T, and use of parts 2 and 3.

Part 2 By induction on the well-formedness derivation of Φ.

Case $\Phi = \bullet$ $\triangleright$ Trivial.

Case $\Phi = \Phi', t$ $\triangleright$ Using part 3 we get the desired result.

Case $\Phi = \Phi', X_i$ $\triangleright$
We have that $\mathsf{unspec}_\Psi,\ \widehat{\Psi}_u \sqsubseteq \mathsf{relevant}\,(\Psi,\ \Psi_u \vdash \Phi'\ \mathrm{wf}) \circ ((\Psi,\ \Psi_u)\,\widehat{@}\,i)$.
We split cases based on whether $i < |\Psi|$ or not.
In the first case:
We trivially have $\mathsf{unspec}_\Psi,\ \widehat{\Psi}_u \sqsubseteq \mathsf{relevant}\,(\Psi,\ \Psi_u \vdash \Phi'\ \mathrm{wf})$, thus directly by use of the induction hypothesis and the same rule for relevancy we get the desired.

In the second case:
Assume without loss of generality $\widehat{\Psi}'_u$ such that $\mathsf{unspec}_\Psi,\ \widehat{\Psi}'_u \sqsubseteq \mathsf{relevant}\,(\Psi,\ \Psi_u \vdash \Phi'\ \mathrm{wf})$, and $(\mathsf{unspec}_\Psi,\ \widehat{\Psi}_u) = (\mathsf{unspec}_\Psi,\ \widehat{\Psi}'_u) \circ ((\Psi,\ \Psi_u)\,\widehat{@}\,i)$.
Then by induction hypothesis get that $\mathsf{unspec}_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash \Phi' \cdot \sigma'_\Psi\ \mathrm{wf})$.
Now we have that $(\Phi',\ X_i) \cdot \sigma'_\Psi = \Phi' \cdot \sigma'_\Psi,\ X_{i-|\Psi|+|\Psi'|}$.
Thus $\mathsf{relevant}\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash (\Phi',\ X_i) \cdot \sigma'_\Psi\ \mathrm{wf}) = \mathsf{relevant}\left(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash (\Phi' \cdot \sigma'_\Psi,\ X_{i-|\Psi|+|\Psi'|})\ \mathrm{wf}\right) = \mathsf{relevant}\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash \Phi' \cdot \sigma'_\Psi\ \mathrm{wf}) \circ ((\Psi',\ \Psi_u \cdot \sigma_\Psi)\,\widehat{@}\,i - |\Psi| + |\Psi'|)$.
Thus we have that $(\mathsf{unspec}_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi) \circ ((\Psi',\ \Psi_u \cdot \sigma_\Psi)\,\widehat{@}\,i - |\Psi| + |\Psi'|) \sqsubseteq \mathsf{relevant}\,(\Psi',\ \Psi_u \cdot \sigma_\Psi \vdash (\Phi',\ X_i) \cdot \sigma'_\Psi\ \mathrm{wf})$.
But $(\mathsf{unspec}_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi) \circ ((\Psi',\ \Psi_u \cdot \sigma_\Psi)\,\widehat{@}\,i - |\Psi| + |\Psi'|) = \mathsf{unspec}_{\Psi'},\ \widehat{\Psi}_u \cdot \sigma_\Psi$.
This is because $(\mathsf{unspec}_\Psi,\ \widehat{\Psi}_u) = (\mathsf{unspec}_\Psi,\ \widehat{\Psi}'_u) \circ ((\Psi,\ \Psi_u)\,\widehat{@}\,i)$, so the i-th element is the only one where $\mathsf{unspec}_\Psi, \widehat{\Psi}'_u$ might differ from $\mathsf{unspec}_\Psi, \widehat{\Psi}_u$; this will be the $i - |\Psi| + |\Psi'|$-th element after σ'_Ψ is applied; and that element is definitely equal after the join.

Part 3 By induction on the typing derivation for t.

Case $t = c, s,$ or $v_{\mathbf{I}}$ $\triangleright$ Trivial using part 2.

Case $t = \Pi(t_1).t_2$ $\triangleright$ By induction hypothesis for $\lceil t_2 \rceil$.

Case $t = \lambda(t_1).t_2$ $\triangleright$ By induction hypothesis for $\lceil t_2 \rceil$.

Case $t = t_1\ t_2$ $\triangleright$ Assume $\widehat{\Psi}_1$ and $\widehat{\Psi}_2$ such that $\widehat{\Psi} = \widehat{\Psi}_1 \circ \widehat{\Psi}_2$. Then use induction hypothesis for t_1 and t_2. Last combine the results using join to get the desired, noticing that both $\widehat{\Psi}_1 \cdot \sigma_\Psi$ and $\widehat{\Psi}_2 \cdot \sigma_\Psi$ are $\sqsubseteq \widehat{\Psi} \cdot \sigma_\Psi$ (so join is defined between them), and also that $(\widehat{\Psi}_1 \circ \widehat{\Psi}_2) \cdot \sigma_\Psi = \widehat{\Psi}_1 \cdot \sigma_\Psi \circ \widehat{\Psi}_2 \circ \sigma_\Psi$.

Case $t = X_i/\sigma$ $\triangleright$
We split cases based on whether $i < |\Psi|$ or not. In case it is, the proof is trivial using part 4. We thus focus on the case where $i \geq |\Psi|$.
We have that $\mathsf{unspec}_\Psi,\ \widehat{\Psi} \sqsubseteq \mathsf{relevant}\,((\Psi, \Psi_u)\lfloor_i \vdash [\Phi']t : [\Phi']s) \circ \mathsf{relevant}\,(\Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi') \circ ((\Psi,\ \Psi_u)\,\widehat{@}\,i)$.
Assume $\widehat{\Psi}^1_u, \widehat{\Psi}'_u, \widehat{\Psi}^2_u$ such that $(\widehat{\Psi}^1_u = \widehat{\Psi}'_u,\ \overbrace{?, \cdots, ?}^{|\Psi|+|\Psi_u|-i \text{ times}})$, $\mathsf{unspec}_\Psi,\ \widehat{\Psi}'_u \sqsubseteq \mathsf{relevant}\,((\Psi, \Psi_u)\lfloor_i \vdash [\Phi']t : [\Phi']s)$, $\mathsf{unspec}_\Psi,\ \widehat{\Psi}^2_u \sqsubseteq \mathsf{relevant}\,(\Psi,\ \Psi_u;\ \Phi \vdash \sigma : \Phi')$ and last that $\widehat{\Psi} = \widehat{\Psi}^1_u \circ \widehat{\Psi}^2_u \circ ((\Psi,\ \Psi_u\,\widehat{@}\,i))$.
By induction hypothesis for $[\Phi']t$ we get that:
$\mathsf{unspec}_{\Psi'},\ \widehat{\Psi}'_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}\,((\Psi', \Psi_u \cdot \sigma_\Psi)\lfloor_i \vdash [\Phi' \cdot \sigma'_\Psi]t \cdot \sigma'_\Psi : [\Phi' \cdot \sigma'_\Psi]s \cdot \sigma'_\Psi)$.
By induction hypothesis for σ we get that:
$\mathsf{unspec}_{\Psi'},\ \widehat{\Psi}^2_u \cdot \sigma_\Psi \sqsubseteq \mathsf{relevant}\,(\Psi',\ \Psi_u \cdot \sigma_\Psi;\ \Phi \cdot \sigma'_\Psi \vdash \sigma \cdot \sigma'_\Psi : \Phi' \cdot \sigma'_\Psi)$
We combine the above to get the desired, using the properties of join at $\widehat{@}$ as we did earlier.

Case (rest) $\triangleright$ Similar to the above cases.

Part 4 Similar as above.

D.4 Unification

Here, we are matching a term with some unification variables against a closed term. Therefore we will use typing judgements like $\Psi \vdash_p T : K$ instead of $\Psi',\ \Psi_u \vdash_p T : K$, as we did above. The single Ψ that we use actually corresponds to Ψ_u; the normal context Ψ' is empty.

First, we need to define the notion of partial substitutions, corresponding to substitutions for partial contexts as defined above.

Definition D.19 (Partial substitutions) *The syntax for partial substitutions follows.*

$\widehat{\sigma_\Psi} ::= \bullet \mid \widehat{\sigma_\Psi},\ T \mid \widehat{\sigma_\Psi},\ ?$

Definition D.20 *Joining two partial substitutions is defined below.*

$\boxed{\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'}$

$$\begin{array}{lcl}
\bullet \circ \bullet & = & \bullet \\
(\widehat{\sigma_\Psi},\ T) \circ (\widehat{\sigma_\Psi}',\ T) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ T \\
(\widehat{\sigma_\Psi},\ ?) \circ (\widehat{\sigma_\Psi}',\ T) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ T \\
(\widehat{\sigma_\Psi},\ T) \circ (\widehat{\sigma_\Psi}',\ ?) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ T \\
(\widehat{\sigma_\Psi},\ ?) \circ (\widehat{\sigma_\Psi}',\ ?) & = & (\widehat{\sigma_\Psi} \circ \widehat{\sigma_\Psi}'),\ ?
\end{array}$$

Definition D.21 *Comparing two partial substitutions is defined below.*

$\boxed{\widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}'}$

$$\begin{array}{lcl}
\bullet \sqsubseteq \bullet & & \\
(\widehat{\sigma_\Psi},\ T) \sqsubseteq (\widehat{\sigma_\Psi}',\ T) & \Leftarrow & \widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}' \\
(\widehat{\sigma_\Psi},\ ?) \sqsubseteq (\widehat{\sigma_\Psi}',\ T) & \Leftarrow & \widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}' \\
(\widehat{\sigma_\Psi},\ ?) \sqsubseteq (\widehat{\sigma_\Psi}',\ ?) & \Leftarrow & \widehat{\sigma_\Psi} \sqsubseteq \widehat{\sigma_\Psi}'
\end{array}$$

Definition D.22 *The fully unspecified substitution for a specific partial context is defined as:*

$\boxed{\mathsf{unspec}_{\widehat{\Psi}} = \widehat{\sigma_\Psi}}$

$$\begin{array}{lcl}
\mathsf{unspec}_{\bullet} & = & ? \\
\mathsf{unspec}_{\widehat{\Psi},\ ?} & = & \mathsf{unspec}_{\widehat{\Psi}},\ ? \\
\mathsf{unspec}_{\widehat{\Psi},\ K} & = & \mathsf{unspec}_{\widehat{\Psi}},\ ?
\end{array}$$

Definition D.23 *Applying a partial extension substitution to a term, a context, or a substitution is entirely identical to normal substitution. It fails when a metavariable that is left unspecified in the extension substitution gets used, something that already happens from the existing definition B.77.*

Definition D.24 *Replacing an unspecified element of a partial substitution with another works as follows.*

$\boxed{\widehat{\sigma_\Psi}[i \mapsto T] = \widehat{\sigma_\Psi}'}$

$$\begin{array}{lcl}
(\widehat{\sigma_\Psi},\ ?)[i \mapsto T] & = & \widehat{\sigma_\Psi},\ T \text{ when } i = |\widehat{\sigma_\Psi}| \\
(\widehat{\sigma_\Psi},\ T')[i \mapsto T] & = & \widehat{\sigma_\Psi}[i \mapsto T],\ T' \text{ when } i < |\widehat{\sigma_\Psi}| \\
(\widehat{\sigma_\Psi},\ ?)[i \mapsto T] & = & \widehat{\sigma_\Psi}[i \mapsto T],\ ? \text{ when } i < |\widehat{\sigma_\Psi}|
\end{array}$$

Definition D.25 *Limiting a partial substitution to a specific partial context works as follows; we assume* $|\widehat{\sigma_\Psi}| = |\widehat{\Psi}|$.

$\boxed{\widehat{\sigma_\Psi}|_{\widehat{\Psi}}}$

$$\begin{array}{lcl}
(\bullet)|_\bullet & = & \bullet \\
(\widehat{\sigma_\Psi},\ T)|_{\widehat{\Psi},\ ?} & = & \widehat{\sigma_\Psi}|_{\widehat{\Psi}},\ ? \\
(\widehat{\sigma_\Psi},\ T)|_{\widehat{\Psi},\ K} & = & \widehat{\sigma_\Psi}|_{\widehat{\Psi}},\ T \\
(\widehat{\sigma_\Psi},\ ?)|_{\widehat{\Psi},\ ?} & = & \widehat{\sigma_\Psi}|_{\widehat{\Psi}},\ ?
\end{array}$$

Definition D.26 *Typing for partial substitutions is defined below.*

$\boxed{\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}}$

$$\frac{}{\bullet \vdash \bullet : \bullet} \qquad \frac{\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi} \qquad \bullet \vdash T : K \cdot \widehat{\sigma_\Psi}}{\bullet \vdash_p (\widehat{\sigma_\Psi},\ T) : (\widehat{\Psi},\ K)} \qquad \frac{\bullet \vdash_p \widehat{\sigma_\Psi} : \widehat{\Psi}}{\bullet \vdash_p (\widehat{\sigma_\Psi},\ ?) : (\widehat{\Psi},\ ?)}$$

Lemma D.27 *If* $\bullet \vdash \widehat{\sigma_\Psi}_1 : \widehat{\Psi}_1$ *and* $\bullet \vdash \widehat{\sigma_\Psi}_2 : \widehat{\Psi}_2$*, with* $\widehat{\Psi}_1 \circ \widehat{\Psi}_2$ *and* $\widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2$ *defined, then* $\bullet \vdash \widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2 : \widehat{\Psi}_1 \circ \widehat{\Psi}_2$.

By induction on the derivation of $\widehat{\sigma_\Psi}_1 \circ \widehat{\sigma_\Psi}_2 = \widehat{\sigma_\Psi}'$.

Case $\bullet \circ \bullet \ \triangleright$ Trivial, since $\widehat{\Psi}_1 = \widehat{\Psi}_2 = \bullet$ by typing inversion.

Case $(\widehat{\sigma_\Psi}'_1,\ T) \circ (\widehat{\sigma_\Psi}'_2,\ T) \ \triangleright$ By typing inversion get $\widehat{\Psi}_1 = \widehat{\Psi}'_1,\ K$ with $T : K$, and $\widehat{\Psi}_2 = \widehat{\Psi}'_2,\ K$ with $T : K$. Thus $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 = \widehat{\Psi}'_1 \circ \widehat{\Psi}'_2,\ K$, and by induction hypothesis for $\widehat{\sigma_\Psi}'_1, \widehat{\sigma_\Psi}'_2$ and typing it is easy to prove the desired.

Case $\dfrac{(\widehat{\sigma_\Psi}'_1,\ ?) \circ (\widehat{\sigma_\Psi}'_2,\ T)}{B} \ \triangleright$ y typing inversion get $\widehat{\Psi}_1 = \widehat{\Psi}'_1,\ ?$, and $\widehat{\Psi}_2 = \widehat{\Psi}'_2,\ K$ with $T : K$. Thus $\widehat{\Psi}_1 \circ \widehat{\Psi}_2 = \widehat{\Psi}'_1 \circ \widehat{\Psi}'_2,\ K$, and by induction hypothesis for $\widehat{\sigma_\Psi}'_1, \widehat{\sigma_\Psi}'_2$ and typing it is easy to prove the desired.

Case $\dfrac{(\widehat{\sigma_\Psi}'_1,\ T) \circ (\widehat{\sigma_\Psi}'_2,\ ?)}{S} \ \triangleright$ imilar to the above.

Case $\dfrac{(\widehat{\sigma_\Psi}'_1,\ ?) \circ (\widehat{\sigma_\Psi}'_2,\ ?)}{A} \ \triangleright$ gain by induction hypothesis and the fact that $\widehat{\Psi}_1 = \widehat{\Psi}'_1, ?$ and $\widehat{\Psi}_2 = \widehat{\Psi}'_2, ?$ by typing inversion.

Lemma D.28 *If* $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$*,* $\bullet \vdash \widehat{\Psi}'$ *wf and* $\widehat{\Psi}' \sqsubseteq \widehat{\Psi}$*, then* $\widehat{\sigma_\Psi}|_{\widehat{\Psi}'} \sqsubseteq \widehat{\sigma_\Psi}$ *and* $\bullet \vdash \widehat{\sigma_\Psi}|_{\widehat{\Psi}'} : \widehat{\Psi}'$.

Trivial by induction on the derivation of $\widehat{\sigma_\Psi}|_{\widehat{\Psi}'}$.

Now we are ready to proceed to a proof about the fact that either a unique unification partial substitution exists for patterns and terms that are typed under the restrictive typing, or that no such substitution exists. The constructive content of this proof will be our unification procedure.

To prove the following theorem we assume that if $\Psi;\ \Phi \vdash_p t : t'$, with $t' \neq \mathsf{Type}'$, the derivation $\Psi;\ \Phi \vdash_p t' : s$ for a suitable s is a sub-derivation of the derivation $\Psi;\ \Phi \vdash_p t : t'$. The way we have written our rules this is actually not true, but an adaptation where the $t' : s$ derivation becomes part of the $t : t'$ derivation is possible, thanks to the theorem B.68.

Theorem D.29 (Decidability and determinism of unification) *1. If $\Psi \vdash_p \Phi$ wf, $\bullet \vdash_p \Phi'$ wf, $\mathsf{relevant}(\Psi \vdash_p \Phi\ wf) = \widehat{\Psi}$, then there either exists a unique substitution $\widehat{\sigma_\Psi}$ such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$ and $\Phi \cdot \widehat{\sigma_\Psi} = \Phi'$, or no such substitution exists.*

2. If $\Psi;\ \Phi \vdash_p t : t_T$, $\bullet;\ \Phi' \vdash_p t' : t'_T$ and $\mathsf{relevant}(\Psi;\ \Phi \vdash t : t'_T) = \widehat{\Psi'}$, then:
assuming that $\Psi;\ \Phi \vdash_p t_T : s$, $\bullet;\ \Phi \vdash_p t'_T : s$, $\mathsf{relevant}(\Psi;\ \Phi \vdash_p t_T : s) = \widehat{\Psi}$ (or, if $t_T = \mathsf{Type}'$, that $\Psi \vdash_p \Phi$ wf, $\bullet \vdash_p \Phi$ wf, $\mathsf{relevant}(\Psi \vdash_p \Phi\ wf) = \widehat{\Psi}$) and there exists a unique substitution $\widehat{\sigma_\Psi}$ such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$, $\Phi \cdot \widehat{\sigma_\Psi} = \Phi'$ and $t_T \cdot \widehat{\sigma_\Psi} = t'_T$,
then there either exists a unique substitution $\widehat{\sigma_\Psi}'$ such that $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi'}$, $\Phi \cdot \widehat{\sigma_\Psi}' = \Phi'$, $t_T \cdot \widehat{\sigma_\Psi}' = t'_T$ and $t \cdot \widehat{\sigma_\Psi}' = t'$, or no such substitution exists.

3. If $\Psi \vdash_p T : K$, $\bullet \vdash_p T' : K$ and $\mathsf{relevant}(\Psi;\ \Phi \vdash T : K) = \Psi$, then either there exists a unique substitution σ_Ψ such that $\bullet \vdash \sigma_\Psi : \Psi$ and $T \cdot \sigma_\Psi = T'$, or no such substitution exists.

Part 2 By induction on the typing derivation for t.

Case $\dfrac{c : t \in \Sigma}{\Psi;\ \Phi \vdash_p c : t_T}\ \triangleright$

We have $t \cdot \widehat{\sigma_\Psi}' = c \cdot \widehat{\sigma_\Psi}' = c$. So for any substitution to satisfy the desired properties we need to have that $t' = c$ also; if this isn't so, no $\widehat{\sigma_\Psi}'$ possibly exists. If we have that $t = t' = c$, then the desired is proved directly by assumption, considering that $\mathsf{relevant}(\Psi;\ \Phi \vdash_p c : t) = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_T : s) = \mathsf{relevant}(\Psi \vdash_p \Phi\ \mathrm{wf})$ (since t_T comes from the definitions context and can therefore not contain extension variables).

Case $\dfrac{\Phi.\mathbf{I} = t}{\Psi;\ \Phi \vdash_p f_\mathbf{I} : t_T}\ \triangleright$

Similarly as above. First, we need $t' = f_{\mathbf{I}'}$, otherwise no suitable $\widehat{\sigma_\Psi}'$ exists. From assumption we have a unique $\widehat{\sigma_\Psi}$ for $\mathsf{relevant}(\Psi;\ \Phi \vdash_p t_T : s)$. If $\mathbf{I} \cdot \widehat{\sigma_\Psi} = \mathbf{I}'$, then $\widehat{\sigma_\Psi}$ has all the desired properties for $\widehat{\sigma_\Psi}'$, considering the fact that $\mathsf{relevant}(\Psi;\ \Phi \vdash_p f_\mathbf{I} : t_T) = \mathsf{relevant}(\Psi \vdash_p \Phi\ \mathrm{wf})$ and $\mathsf{relevant}(\Psi;\ \Phi \vdash_p t_T : s) = \mathsf{relevant}(\Psi \vdash_p \Phi\ \mathrm{wf})$ (since $t_T = \Phi.i$). It is also unique, because an alternate $\widehat{\sigma_\Psi}'$ would violate the assumed uniqueness of $\widehat{\sigma_\Psi}$. If $\mathbf{I} \cdot \widehat{\sigma_\Psi} \neq \widehat{\sigma_\Psi}'$, no suitable substitution exists, because of the same reason.

Case $\dfrac{(s, s') \in \mathcal{A}}{\Psi;\ \Phi \vdash_p s : s'}\ \triangleright$

Entirely similar to the case for c.

Case $\dfrac{\Psi;\ \Phi \vdash_p t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi;\ \Phi \vdash_p \Pi(t_1).t_2 : s''}\ \triangleright$

First, we have either that $t' = \Pi(t'_1).t'_2$, or no suitable $\widehat{\sigma_\Psi}'$ exists. Thus by inversion for t' we get:
$\bullet;\ \Phi' \vdash_p t'_1 : s_*$, $\bullet;\ \Phi',\ t'_1 \vdash_p \lceil t'_2 \rceil_{|\Phi'|} : s'_*$, $(s_*, s'_*, s'') \in \mathcal{R}$.
Now, we need $s = s_*$, otherwise no suitable $\widehat{\sigma_\Psi}'$ possibly exists. To see why this is so, assume that a $\widehat{\sigma_\Psi}'$ satisfying the necessary conditions exists, and $s \neq s_*$; then we have that $t_1 \cdot \widehat{\sigma_\Psi}' = t'_1$, which means that their types should also match, a contradiction.
We use the induction hypothesis for t_1 and t'_1. We are allowed to do so because $\mathsf{relevant}(\Psi;\ \Phi \vdash_p s'' : s''') = \mathsf{relevant}(\Psi;\ \Phi \vdash_p s : s'''')$, and the other properties for $\widehat{\sigma_\Psi}$ also hold trivially.

From that we either get a $\widehat{\sigma_\Psi}'$ such that: $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$, where $\widehat{\Psi}' = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_1 : s)$ and $t_1 \cdot \widehat{\sigma_\Psi}' = t_1'$, $\Phi \cdot \widehat{\sigma_\Psi}' = \Phi'$. Since a partial substitution unifying t with t' will also include a substitution that only has to do with $\widehat{\Psi}'$, we see that if no $\widehat{\sigma_\Psi}'$ is returned by the induction hypothesis, no suitable substitution for t and t' actually exists.
We can now use the induction hypothesis for t_2 and $\widehat{\sigma_\Psi}'$, since $\mathsf{relevant}(\Psi;\ \Phi\, t_1 \vdash_p s' : s'''') = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_1 : s)$, and the other requirements trivially hold. Especially for s' and s'_* being equal, this is trivial since both need to be equal to s'' (because of the form of our rule set $\mathcal{R}$).
From that we either get a $\widehat{\sigma_\Psi}''$ such that, $\bullet \vdash \widehat{\sigma_\Psi}'' : \widehat{\Psi}''$, $\lceil t_2 \rceil_{|\Phi|} \cdot \widehat{\sigma_\Psi}'' = \lceil t_2' \rceil_{|\Phi'|}$, $\Phi \cdot \widehat{\sigma_\Psi}'' = \Phi$ and $t_1 \cdot \widehat{\sigma_\Psi}'' = t_1'$, or that such $\widehat{\sigma_\Psi}''$ does not exist. In the second case we proceed as above, so we focus in the first case.
By use of properties of freshening (like injectivity) we are led to the fact that $(\Pi(t_1).t_2) \cdot \widehat{\sigma_\Psi}'' = \Pi(t_1').(t_2')$, so the returned $\widehat{\sigma_\Psi}''$ has the desired properties, if we consider the fact that $\mathsf{relevant}(\Psi;\ \Phi \vdash_p \Pi(t_1).t_2 : s'') = \mathsf{relevant}\left(\Psi;\ \Phi, t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : s'\right)$.

Case $\dfrac{\Psi;\ \Phi \vdash_p t_1 : s \qquad \Psi;\ \Phi,\ t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : t_3 \qquad \Psi;\ \Phi \vdash_p \Pi(t_1).\lfloor t_3 \rfloor_{|\Phi|,\,.} : s'}{\Psi;\ \Phi \vdash_p \lambda(t_1).t_2 : \Pi(t_1).\lfloor t_3 \rfloor_{|\Phi|,\,.}}\ \triangleright$

We have that either $t' = \lambda(t_1').t_2'$, or no suitable $\widehat{\sigma_\Psi}'$ exists. Thus by typing inversion for t' we get:
$\bullet;\ \Phi' \vdash_p t_1' : s_*$, $\bullet;\ \Phi',\ t_1' \vdash_p \lceil t_2' \rceil_{|\Phi'|} : t_3'$, $\bullet;\ \Phi' \vdash_p \Pi(t_1').\lfloor t_3 \rfloor_{|\Phi'|,\,.} : s'_*$.
By assumption we have that there exists a unique $\widehat{\sigma_\Psi}$ such that $\mathsf{relevant}\left(\Psi;\ \Phi \vdash_p \Pi(t_1).\lfloor t_3 \rfloor_{|\Phi|,\,.} : s\right) = \widehat{\Psi}$, $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$, $\Phi \cdot \widehat{\sigma_\Psi} = \Phi'$, $(\Pi(t_1).\lfloor t_3 \rfloor) \cdot \widehat{\sigma_\Psi} = \Pi(t_1').\lfloor t_3' \rfloor$, if $\mathsf{relevant}\left(\Psi;\ \Phi \vdash_p \Pi(t_1).\lfloor t_3 \rfloor_{|\Phi,\cdot|}\right) = \widehat{\Psi}$. From that we also get that $s' = s'_*$.
From the fact that $(\Pi(t_1).\lfloor t_3 \rfloor) \cdot \widehat{\sigma_\Psi} = \Pi(t_1').\lfloor t_3' \rfloor$, we get first of all that $t_1 \cdot \widehat{\sigma_\Psi} = t_1'$, and also that $t_3 \cdot \widehat{\sigma_\Psi} = t_3'$.
Furthermore, We have that $\mathsf{relevant}(\Psi;\ \Phi \vdash_p \Pi(t_1).\lfloor t_3 \rfloor : s') = \mathsf{relevant}(\Psi;\ \Phi,\ t_1 \vdash_p t_3 : s')$.
From that we understand that $\widehat{\sigma_\Psi}$ is a suitable substitution to use for the induction hypothesis for $\lceil t_2 \rceil$.
Thus from induction hypothesis we either get a unique $\widehat{\sigma_\Psi}'$ with the properties: $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$, $\lceil t_2 \rceil \cdot \widehat{\sigma_\Psi}' = \lceil t_2' \rceil$, $(\Phi,\ t_1) \cdot \widehat{\sigma_\Psi} = \Phi'$, t_1', $t_3 \cdot \widehat{\sigma_\Psi} = t_3'$, if $\mathsf{relevant}\left(\Psi;\ \Phi,\ t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : t_3\right) = \widehat{\Psi}$, or that no such substitution exists.
We focus on the first case; in the second case no unifying substitution for t and t' exists, otherwise the lack of existence of a suitable $\widehat{\sigma_\Psi}'$ would lead to a contradiction.
This substitution $\widehat{\sigma_\Psi}'$ has the desired properties with respect to unification of t against t' (again using the properties of freshening, like injectivity), and it is unique, because the existence of an alternate substitution with the same properties would violate the uniqueness assumption of the substitution returned by induction hypothesis.

Case $\dfrac{\Psi;\ \Phi \vdash_p t_1 : \Pi(t_a).t_b \qquad \Psi;\ \Phi \vdash_p t_2 : t_a}{\Psi;\ \Phi \vdash_p t_1\ t_2 : \lceil t_b \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}\ \triangleright$

Again we have that either $t' = t_1'\ t_2'$, or no suitable substitution possibly exists. Thus by inversion of typing for t' we get:
$\bullet;\ \Phi \vdash_p t_1' : \Pi(t_a').t_b'$, $\bullet;\ \Phi \vdash_p t_2' : t_a'$, $t_T' = \left\lceil t_b' \right\rceil_{|\Phi'|} \cdot (\mathsf{id}_{\Phi'},\ t_2')$.
Furthermore we have that $\Psi;\ \Phi \vdash_p \Pi(t_a).t_b : s$ and $\bullet;\ \Phi \vdash_p \Pi(t_a').t_b' : s'$ for suitable s, s'. We need $s = s'$, otherwise no suitable $\widehat{\sigma_\Psi}'$ exists (because if t_1 and t_1' were unifiable by substitution, their Π-types would match, and also their sorts, which is a contradiction).
We can use the induction hypothesis for $\Pi(t_a).t_b$ and $\Pi(t_a').t_b'$, with the partial substitution $\widehat{\sigma_\Psi}$ limited only to those variables relevant in $\Psi \vdash_p \Phi$ wf. In that case all of the requirements for $\widehat{\sigma_\Psi}$ hold (the uniqueness condition also holds for this substitution, using part 1 for the fact that Φ and Φ' only have a unique unification substitution), so we get from the induction hypothesis either a $\widehat{\sigma_\Psi}'$ for $\widehat{\Psi}' = \mathsf{relevant}(\Psi;\ \Phi \vdash_p \Pi(t_a).t_b : s)$ such that $\Phi \cdot \widehat{\sigma_\Psi} = \Phi'$ and $(\Pi(t_a).t_b) \cdot \widehat{\sigma_\Psi} = \Pi(t_a').t_b'$, or that no such $\widehat{\sigma_\Psi}'$ exists. In the second case, again we can show

that no suitable substitution for t and t' exists; so we focus in the first case.
We can now use the induction hypothesis for t_1, using this $\widehat{\sigma_\Psi}'$. From that, we get that either a $\widehat{\sigma_{\Psi_1}}$ exists for $\widehat{\Psi}_1 = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_1 : \Pi(t_a).t_b)$ such that $t_1 \cdot \widehat{\sigma_{\Psi_1}} = t_1'$ etc., or that no such $\widehat{\sigma_{\Psi_1}}$ exists, in which case we argue that no global $\widehat{\sigma_\Psi}'$ exists for unifying t and t' (because we could limit it to the $\widehat{\Psi}_1$ variables and yield a contradiction).
We now form $\widehat{\sigma_\Psi}''$ which is the limitation of $\widehat{\sigma_\Psi}'$ to the context $\widehat{\Psi}'' = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_a : s_*)$. For that, we have that $\bullet \vdash_p \widehat{\sigma_\Psi}'' : \widehat{\Psi}''$, $\Phi \cdot \widehat{\sigma_\Psi}'' = \Phi'$ and $t_a \cdot \widehat{\sigma_\Psi}'' = t_a$. Also it is the unique substitution with those properties, otherwise the induction hypothesis for t_a would be violated.
Using $\widehat{\sigma_\Psi}''$ we can allude to the induction hypothesis for t_2, which either yields a substitution $\widehat{\sigma_{\Psi_2}}$ for $\widehat{\Psi}_2 = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_2 : t_a)$, such that $t_2 \cdot \widehat{\sigma_{\Psi_2}} = t_2'$, etc., or that no such substitution exists, which we prove implies no global unifying substitution exists.
Having now the $\widehat{\sigma_{\Psi_1}}$ and $\widehat{\sigma_{\Psi_2}}$ specified above, we consider the substitution $\widehat{\sigma_{\Psi_r}} = \widehat{\sigma_{\Psi_1}} \circ \widehat{\sigma_{\Psi_2}}$. This substitution, if it exists, has the desired properties: we have that $\widehat{\Psi}_r = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_1\ t_2 : \lceil t_b \rceil \cdot (\mathsf{id}_\Phi,\ t_2)) = \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_1 : \Pi(t_a).t_b) \circ \mathsf{relevant}(\Psi;\ \Phi \vdash_p t_2 : t_a)$, and thus $\bullet \vdash \widehat{\sigma_{\Psi_r}} : \widehat{\Psi}_r$. Also, $(t_1\ t_2) \cdot \widehat{\sigma_{\Psi_r}} = t_1'\ t_2'$, $t_T \cdot \widehat{\sigma_{\Psi_r}} = t_T'$ (because $t_b \cdot \widehat{\sigma_{\Psi_r}} = t_b'$ etc.), and $\Phi \cdot \widehat{\sigma_{\Psi_r}} = \Phi'$. It is also unique: if another substitution had the same properties, we could limit it to either the relevant variables for t_1 or t_2 and get a contradiction. Thus this is the desired substitution.
If $\widehat{\sigma_{\Psi_r}}$ does not exist, then no suitable substitution for unifying t and t' exists. This is again because we could limit any potential such substitution to two parts, $\widehat{\sigma_{\Psi_1}}'$ and $\widehat{\sigma_{\Psi_2}}'$ (for $\widehat{\Psi}_1$ and $\widehat{\Psi}_2$ respectively), violating the uniqueness of the substitutions yielded by the induction hypothesis.

Case $\dfrac{\Psi;\ \Phi \vdash_p t_1 : t_a \qquad \Psi;\ \Phi \vdash_p t_2 : t_a \qquad \Psi;\ \Phi \vdash_p t_a : \mathsf{Type}}{\Psi;\ \Phi \vdash_p t_1 = t_2 : \mathsf{Prop}}\ \triangleright$

Similarly as above. First assume that $t' = (t_1' = t_2')$, with $t_1' : t_a'$, $t_2' : t_a'$ and $t_a' : \mathsf{Type}$. Then, by induction hypothesis get a unifying substitution $\widehat{\sigma_\Psi}'$ for t_a and t_a'. Use that $\widehat{\sigma_\Psi}'$ in order to allude to the induction hypothesis for t_1 and t_2 independently, yielding substitutions $\widehat{\sigma_{\Psi_1}}$ and $\widehat{\sigma_{\Psi_2}}$. Last, claim that the globally required substitution must actually be equal to $\widehat{\sigma_{\Psi_1}} \circ \widehat{\sigma_{\Psi_2}}$.

Case $\dfrac{(\Psi).i = T \qquad T = [\Phi_*]t_T \qquad \Psi;\ \Phi \vdash_p \sigma : \Phi_* \qquad \Phi_* \subseteq \Phi \qquad \sigma = \mathsf{id}_{\Phi_*}}{\Psi;\ \Phi \vdash_p X_i/\sigma : t_T \cdot \sigma}\ \triangleright$

We trivially have $t_T \cdot \sigma = t_T$. We split cases depending on whether $\widehat{\sigma_\Psi}.i =?$ or not. If it is unspecified:
We split cases further depending on whether t' uses any variables higher than $|\Phi_* \cdot \widehat{\sigma_\Psi}| - 1$ or not. That is, if $t' <^f |\Phi_* \cdot \widehat{\sigma_\Psi}|$ or not. In the case where this doesn't hold, it is obvious that there is no possible $\widehat{\sigma_\Psi}'$ such that $(X_i/\sigma) \cdot \widehat{\sigma_\Psi}' = t'$, since $\widehat{\sigma_\Psi}'$ must include $\widehat{\sigma_\Psi}$, and the term $(X_i/\sigma) \cdot \widehat{\sigma_\Psi}'$ can therefore not include variables outside the prefix $\Phi_* \cdot \widehat{\sigma_\Psi}$ of $\Phi \cdot \widehat{\sigma_\Psi}$.
In the case where $t' <^f |\Phi_* \cdot \widehat{\sigma_\Psi}|$, we consider the substitution $\widehat{\sigma_\Psi}' = \widehat{\sigma_\Psi}[i \mapsto t']$. In that case we obviously have $\Phi \cdot \widehat{\sigma_\Psi}' = \Phi'$, $t_T \cdot \widehat{\sigma_\Psi}' = t_T$, and also $t \cdot \widehat{\sigma_\Psi} = t'$. Also, $\widehat{\Psi}' = \mathsf{relevant}(\Psi;\ \Phi \vdash_p X_i/\sigma : t_T \cdot \sigma) = (\mathsf{relevant}(\Psi\lfloor_i;\ \Phi_* \vdash_p t_T : s), ?, \cdots, ?) \circ \mathsf{relevant}(\Psi;\ \Phi \vdash_p \sigma : \Phi_*) \circ (\Psi \widehat{@} i)$.
We need to show that $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$. First, we have that $\mathsf{relevant}(\Psi;\ \Phi \vdash_p \sigma : \Phi^*) = \mathsf{relevant}(\Psi \vdash_p \Phi\ \mathrm{wf})$ since $\Phi^* \subseteq \Phi$. Second, we have that $\mathsf{relevant}(\Psi;\ \Phi \vdash_p t_T : s) = (\mathsf{relevant}(\Psi\lfloor_i;\ \Phi_* \vdash_p t_T : s), ?, \cdots, ?) \circ \mathsf{relevant}(\Psi \vdash_p \Phi\ \mathrm{wf})$. Thus we have that $\widehat{\Psi}' = \widehat{\Psi} \circ (\Psi \widehat{@} i)$. It is now trivial to see that indeed $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$.
If $\widehat{\sigma_\Psi}.i = t_*$, then we split cases on whether $t_* = t'$ or not. If it is, then obviously $\widehat{\sigma_\Psi}$ is the desired unifying substitution for which all the desired properties hold. If it is not, then no substitution with the desired properties possibly exists, because it would violate the uniqueness assumption for $\widehat{\sigma_\Psi}$.

Case *(rest)* $\triangleright$ Similar techniques as above.

Part 1 By induction on the well-formedness derivation for Φ.

Case $\dfrac{}{\Psi \vdash_p \bullet \text{ wf}}$ $\triangleright$
Trivially, we either have $\Phi' = \bullet$, in which case unspec_Ψ is the unique substitution with the desired properties, or no substitution possibly exists.

Case $\dfrac{\Psi \vdash_p \Phi \text{ wf} \qquad \Psi;\, \Phi \vdash_p t : s}{\Psi \vdash_p (\Phi,\, t) \text{ wf}}$ $\triangleright$
We either have that $\Phi' = \Phi',\, t'$ or no substitution possibly exists. By induction hypothesis get $\widehat{\sigma_\Psi}$ such that $\Phi \cdot \widehat{\sigma_\Psi} = \Phi'$ and $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$ with $\widehat{\Psi} = \mathsf{relevant}\,(\Psi \vdash_p \Phi \text{ wf})$. Now we use part 2 to either get a $\widehat{\sigma_\Psi}'$ which is obviously the substitution that we want, since $(\Phi,\, t) \cdot \widehat{\sigma_\Psi}' = \Phi',\, t'$ and $\mathsf{relevant}\,(\Psi \vdash_p (\Phi,\, t) \text{ wf}) = \mathsf{relevant}\,(\Psi;\, \Phi \vdash_p t : s)$; or we get the fact that no such substitution possibly exists. In that case, we again conclude that no substitution for the current case exists either, otherwise it would violate the induction hypothesis.

Case $\dfrac{\bullet \vdash_p \Phi \text{ wf} \qquad (\Psi).i = [\Phi]\,\mathsf{ctx}}{\Psi \vdash_p \Phi,\, X_i \text{ wf}}$ $\triangleright$
We either have $\Phi' = \Phi,\, \Phi''$, or no substitution possibly exists (since Φ does not depend on unification variables, so we always have $\Phi \cdot \widehat{\sigma_\Psi} = \Phi$). We now consider the substitution $\widehat{\sigma_\Psi} = \mathsf{unspec}_\Psi[i \mapsto [\Phi]\Phi'']$. We obviously have that $(\Phi,\, X_i) \cdot \widehat{\sigma_\Psi} = \Phi,\, \Phi''$, and also that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$ with $\widehat{\Psi} = \Psi \widehat{@} i = \mathsf{relevant}\,(\Psi \vdash_p \Phi,\, X_i \text{ wf})$. Thus this substitution has the desired properties.

Part 3 By induction on the typing for T.

Case $\dfrac{\Psi;\, \Phi \vdash_p t : t_T \qquad \Psi;\, \Phi \vdash t_T : s}{\Psi \vdash_p [\Phi]t : [\Phi]t_T}$ $\triangleright$
By inversion of typing for T' we have: $T' = [\Phi]t',\, \bullet;\, \Phi \vdash_p t' : t_T,\, \bullet;\, \Phi \vdash_p t_T : s$.
We obviously have $\widehat{\Psi} = \mathsf{relevant}\,(\Psi;\, \Phi \vdash_p t_T : s) = \mathsf{unspec}_\Psi$, and the substitution $\widehat{\sigma_\Psi} = \mathsf{unspec}_\Psi$ is the unique substitution such that $\bullet \vdash \widehat{\sigma_\Psi} : \widehat{\Psi}$, $\Phi \cdot \widehat{\sigma_\Psi} = \Phi$ and $t_T \cdot \widehat{\sigma_\Psi} = t_T$. We can thus use part 2 for attempting unification between t and t', yielding a $\widehat{\sigma_\Psi}'$ such that $\bullet \vdash \widehat{\sigma_\Psi}' : \widehat{\Psi}'$ with $\widehat{\Psi}' = \mathsf{relevant}\,(\Psi;\, \Phi \vdash_p t : t_T)$ and $t \cdot \widehat{\sigma_\Psi}' = t'$. We have that $\mathsf{relevant}\,(\Psi;\, \Phi \vdash_p t : t_T) = \mathsf{relevant}\,(\Psi \vdash_p T : K)$, thus $\widehat{\Psi}' = \Psi$ by assumption. From that we realize that $\widehat{\sigma_\Psi}'$ is a fully-specified substitution since $\bullet \vdash \widehat{\sigma_\Psi}' : \Psi$, and thus this is the substitution with the desired properties.
If unification between t and t' fails, it is trivial to see that no substitution with the desired substitution exists, otherwise it would lead directly to a contradiction.

Case $\dfrac{\Psi \vdash_p \Phi,\, \Phi' \text{ wf}}{\Psi \vdash_p [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}}$ $\triangleright$
By inversion of typing for T' we have: $T' = [\Phi]\Phi'',\, \bullet \vdash_p \Phi,\, \Phi'' \text{ wf},\, \bullet \vdash_p \Phi \text{ wf}$. From part 1 we get a $\widehat{\sigma_\Psi}$ unifying $\Phi,\, \Phi'$ and $\Phi,\, \Phi''$, or the fact that no such $\widehat{\sigma_\Psi}$ exists. In the first case, as above, it is easy to see that this is the fully-specified substitution that we desire. In the second case, no suitable substitution exists, otherwise we are led directly to a contradiction.

The above proof is constructive. Its computational content is actually a unification algorithm for our patterns. We illustrate the algorithm below by giving its unification rules; notice that it follows the inductive

structure of the proof (and makes the same assumption about types-of-types being subderivations). If a derivation according to the following rules is not possible, the algorithm returns failure.

Definition D.30 (Unification algorithm) *We give the rules for the unification algorithm below.*

$\boxed{(\Psi \vdash_p T : K) \sim (\bullet \vdash_p T' : K) \triangleright \widehat{\sigma_\Psi}}$

$$\frac{(\Psi;\, \Phi \vdash_p t : t_T) \sim \left(\bullet;\, \Phi \vdash_p t' : t_T\right) \triangleleft \mathsf{unspec}_\Psi \triangleright \widehat{\sigma_\Psi}}{(\Psi \vdash_p [\Phi]t : [\Phi]t_T) \sim \left(\bullet \vdash_p [\Phi]t' : [\Phi]t'_T\right) \triangleright \widehat{\sigma_\Psi}} \qquad \frac{\left(\Psi \vdash_p \Phi,\, \Phi' \text{ wf}\right) \sim \left(\bullet \vdash_p \Phi,\, \Phi'' \text{ wf}\right) \triangleright \widehat{\sigma_\Psi}}{\left(\Psi \vdash_p [\Phi]\Phi' : [\Phi]\,\mathsf{ctx}\right) \sim \left(\bullet \vdash_p [\Phi]\Phi'' : [\Phi]\,\mathsf{ctx}\right) \triangleright \widehat{\sigma_\Psi}}$$

$\boxed{(\Psi \vdash_p \Phi \text{ wf}) \sim (\bullet \vdash \Phi' \text{ wf}) \triangleright \widehat{\sigma_\Psi}}$

$$(\Psi \vdash_p \bullet \text{ wf}) \sim (\bullet \vdash \bullet \text{ wf}) \triangleright \mathsf{unspec}_\Psi$$

$$\frac{(\Psi \vdash_p \Phi \text{ wf}) \sim \left(\bullet \vdash_p \Phi' \text{ wf}\right) \triangleright \widehat{\sigma_\Psi} \qquad (\Psi;\, \Phi \vdash_p t : s) \sim \left(\bullet;\, \Phi' \vdash t' : s\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}{(\Psi \vdash_p (\Phi,\, t) \text{ wf}) \sim \left(\bullet \vdash (\Phi',\, t') \text{ wf}\right) \triangleright \widehat{\sigma_\Psi}'}$$

$$(\Psi \vdash_p \Phi,\, X_i \text{ wf}) \sim \left(\bullet \vdash \Phi,\, \Phi' \text{ wf}\right) \triangleright \mathsf{unspec}_\Psi[i \mapsto \Phi']$$

$\boxed{(\Psi;\, \Phi \vdash_p t : t_T) \sim (\bullet;\, \Phi' \vdash t' : t'_T) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}$

$$(\Psi;\, \Phi \vdash_p c : t) \sim \left(\bullet;\, \Phi' \vdash c : t'\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \qquad \left(\Psi;\, \Phi \vdash_p s : s'\right) \sim \left(\bullet;\, \Phi' \vdash s : s'\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'$$

$$\frac{\mathbf{I} \cdot \widehat{\sigma_\Psi} = \mathbf{I}'}{(\Psi;\, \Phi \vdash_p f_{\mathbf{I}} : t) \sim \left(\bullet;\, \Phi' \vdash f_{\mathbf{I}'} : t'\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}}$$

$$\frac{\begin{array}{c}(\Psi;\, \Phi \vdash_p t_1 : s) \sim \left(\bullet;\, \Phi' \vdash t'_1 : s\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \\ \left(\Psi;\, \Phi,\, t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : s'\right) \sim \left(\bullet;\, \Phi',\, t'_1 \vdash_p \lceil t'_2 \rceil_{|\Phi'|} : s'\right) \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_\Psi}''\end{array}}{\left(\dfrac{\Psi;\, \Phi \vdash_p t_1 : s \qquad \Psi;\, \Phi,\, t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\Psi;\, \Phi \vdash_p \Pi(t_1).t_2 : s''}\right) \sim \left(\dfrac{\bullet;\, \Phi \vdash t'_1 : s \qquad \bullet;\, \Phi',\, t'_1 \vdash \lceil t'_2 \rceil_{|\Phi'|} : s' \qquad (s, s', s'') \in \mathcal{R}}{\bullet;\, \Phi' \vdash \Pi(t'_1).t'_2 : s''}\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}''}$$

$$\frac{\left(\Psi;\, \Phi,\, t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : t'\right) \sim \left(\bullet;\, \Phi',\, t'_1 \vdash_p \lceil t'_2 \rceil_{|\Phi'|} : t''\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}{\left(\dfrac{\Psi;\, \Phi \vdash_p t_1 : s \qquad \Psi;\, \Phi,\, t_1 \vdash_p \lceil t_2 \rceil_{|\Phi|} : t' \qquad \Psi;\, \Phi \vdash_p \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|,\,.} : s'}{\Psi;\, \Phi \vdash_p \lambda(t_1).t_2 : \Pi(t_1).\, \lfloor t' \rfloor_{|\Phi|,\,.}}\right) \sim \left(\dfrac{\bullet;\, \Phi' \vdash t'_1 : s \qquad \bullet;\, \Phi',\, t'_1 \vdash \lceil t'_2 \rceil_{|\Phi'|} : t'' \qquad \bullet;\, \Phi' \vdash_p \Pi(t'_1).\, \lfloor t'' \rfloor_{|\Phi'|,\,.} : s'}{\bullet;\, \Phi' \vdash \lambda(t'_1).t'_2 : \Pi(t'_1).\, \lfloor t'' \rfloor_{|\Phi'|,\,.}}\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}'}$$

$$\frac{\begin{array}{c}\left(\Psi;\ \Phi \vdash_p \Pi(t_a).t_b : s'\right) \sim \left(\bullet;\ \Phi \vdash_p \Pi(t'_a).t'_b : s'\right) \triangleleft \widehat{\sigma_\Psi}|_{\mathsf{relevant}(\Psi \vdash_p \Phi\ \mathrm{wf})} \triangleright \widehat{\sigma_\Psi}' \\ (\Psi;\ \Phi \vdash_p t_1 : \Pi(t_a).t_b) \sim \left(\bullet;\ \Phi' \vdash_p t'_1 : \Pi(t'_a).t'_b\right) \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_{\Psi 1}} \\ (\Psi;\ \Phi \vdash_p t_2 : t_a) \sim \left(\bullet;\ \Phi' \vdash_p t'_2 : t'_a\right) \triangleleft \widehat{\sigma_\Psi}'|_{\mathsf{relevant}(\Psi;\ \Phi \vdash_p t_a : s)} \triangleright \widehat{\sigma_{\Psi 2}}\end{array}}{\left(\dfrac{\dfrac{\Psi;\ \Phi \vdash_p t_a : s}{\dfrac{\Psi;\ \Phi \vdash_p \Pi(t_a).t_b : s'}{\Psi;\ \Phi \vdash_p t_1 : \Pi(t_a).t_b}} \qquad \Psi;\ \Phi \vdash_p t_2 : t_a}{\Psi;\ \Phi \vdash_p t_1\ t_2 : \lceil t_b \rceil_{|\Phi|} \cdot (\mathsf{id}_\Phi, t_2)}\right) \sim \left(\dfrac{\dfrac{\dfrac{\bullet;\ \Phi' \vdash_p t'_a : s}{\bullet;\ \Phi \vdash_p \Pi(t'_a).t'_b : s}}{\bullet;\ \Phi' \vdash_p t'_1 : \Pi(t'_a).t'_b} \qquad \bullet;\ \Phi' \vdash_p t'_2 : t'_a}{\bullet;\ \Phi' \vdash_p t'_1\ t'_2 : \lceil t'_b \rceil_{|\Phi'|} \cdot (\mathsf{id}_{\Phi'}, t'_2)}\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_{\Psi 1}} \circ \widehat{\sigma_{\Psi 2}}}$$

$$\frac{\begin{array}{c}(\Psi;\ \Phi \vdash_p t : \mathsf{Type}) \sim \left(\bullet;\ \Phi' \vdash_p t' : \mathsf{Type}\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}' \\ (\Psi;\ \Phi \vdash_p t_1 : t) \sim \left(\bullet;\ \Phi' \vdash_p t'_1 : t'\right) \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_{\Psi 1}} \qquad (\Psi;\ \Phi \vdash_p t_2 : t) \sim \left(\bullet;\ \Phi' \vdash_p t'_2 : t'\right) \triangleleft \widehat{\sigma_\Psi}' \triangleright \widehat{\sigma_{\Psi 2}}\end{array}}{\left(\dfrac{\Psi;\ \Phi \vdash_p t_1 : t \qquad \Psi;\ \Phi \vdash_p t_2 : t}{\dfrac{\Psi;\ \Phi \vdash_p t : \mathsf{Type}}{\Psi;\ \Phi \vdash_p t_1 = t_2 : \mathsf{Prop}}}\right) \sim \left(\dfrac{\bullet;\ \Phi \vdash_p t'_1 : t' \qquad \bullet;\ \Phi \vdash_p t'_2 : t'}{\dfrac{\bullet;\ \Phi \vdash_p t' : \mathsf{Type}}{\bullet;\ \Phi \vdash_p t'_1 = t'_2 : \mathsf{Prop}}}\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_{\Psi 1}} \circ \widehat{\sigma_{\Psi 2}}}$$

$$\frac{\widehat{\sigma_\Psi}.i = ? \qquad \Psi.i = [\Phi^*]t_T \qquad t' <^f |\Phi^* \cdot \widehat{\sigma_\Psi}|}{(\Psi;\ \Phi \vdash_p X_i/\sigma : t_T \cdot \sigma) \sim \left(\bullet;\ \Phi' \vdash t' : t'_T\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}[i \mapsto [\Phi^*]t']}$$

$$\frac{\widehat{\sigma_\Psi}.i = [\Phi^*]t \qquad t = t'}{(\Psi;\ \Phi \vdash_p X_i/\sigma : t_T) \sim \left(\bullet;\ \Phi' \vdash t' : t'_T\right) \triangleleft \widehat{\sigma_\Psi} \triangleright \widehat{\sigma_\Psi}}$$

Lemma D.31 *The above rules are algorithmic.*

Proved by the fact that they obey structural induction on the typing derivations, and are deterministic; non-covered cases signify the unification failure result.

By mimicking the unification proof above, we could show independently that the above algorithm is sound – that is, that the $\widehat{\sigma_\Psi}'$ it returns if it is successful is actually a substitution that makes t and t' unify (as well as Φ and Φ', along with t_T and t'_T) and is of the right type, provided that the assumptions about the input substitution $\widehat{\sigma_\Psi}$ do hold. Furthermore, we could show completeness, the fact that if the algorithm fails, no such substitution actually exists.

D.5 Computational language

Here we will refine our results for progress and preservation from the previous section, using the above results.

Definition D.32 *We refine the typing rule for pattern matching from definition C.4 as shown below.*

$$\frac{\begin{array}{c}\Psi \vdash T : K \qquad \Psi,\ K;\ \Gamma \vdash \lceil \tau \rceil_{|\Psi|,1} : \star \qquad \Psi \vdash_p \lceil \Psi' \rceil_{|\Psi|}\ \mathrm{wf} \\ \Psi,\ \lceil \Psi' \rceil_{|\Psi|} \vdash_p \lceil T' \rceil_{|\Psi|,|\Psi'|} : K \qquad \mathsf{unspec}_\Psi,\ \lceil \Psi' \rceil_{|\Psi|} \sqsubseteq \mathsf{relevant}\left(\Psi,\ \lceil \Psi' \rceil_{|\Psi|} \vdash_p \lceil T' \rceil_{|\Psi|,|\Psi'|} : K\right) \\ \Psi,\ \lceil \Psi' \rceil_{|\Psi|};\ \Sigma;\ \Gamma \vdash \lceil e' \rceil_{|\Psi|,|\Psi'|} : \lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\ \lceil T' \rceil_{|\Psi|,|\Psi'|})\end{array}}{\Psi;\ \Sigma;\ \Gamma \vdash \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi'.T' \mapsto e') : (\lceil \tau \rceil_{|\Psi|,1} \cdot (\mathsf{id}_\Psi,\ T)) + \mathsf{unit}}$$

Lemma D.33 (Substitution) *Adaptation of the substitution lemma from C.13.*

All the cases are entirely identical to the previous proof, with the exception of the pattern matching construct which has a new typing rule. In that case, proceed similarly as before, with the use of the lemmas D.2, D.3 and D.18 proved above.

Theorem D.34 (Preservation) *Adaptation of theorem C.16 to the new rules.*

All the cases are entirely identical to the previous proof, with the exception of the pattern matching construct. In that case, we need to explicitly allude to the fact that if $\bullet \vdash_p \lceil \Psi' \rceil_{|\Psi|}$ wf, then obviously also $\bullet \vdash \lceil \Psi' \rceil_{|\Psi|}$ wf. Similarly we have that $\lceil \Psi' \rceil_0 \vdash_p \lceil T' \rceil_{0,|\Psi'|} : K$ implies $\lceil \Psi' \rceil_0 \vdash \lceil T' \rceil_{0,|\Psi'|} : K$.

Theorem D.35 (Progress) *Adaptation of theorem E.11 to the new rules.*

Again the only case that needs adaptation is the pattern matching case. In that case, we first note that if $\bullet \vdash T : K$ (as we have here), we also have $\bullet \vdash_p T : K$. Then, we allude to the theorem 3 to split cases depending on whether a suitable σ_Ψ exists or not. In both cases, one step rule is applicable – if a unique σ_Ψ exists, then it has the desired properties for the first pattern matching step rule to work; if it does not, the second pattern matching step rule is applicable.

D.6 Sketch: practical pattern matching

The unification algorithm presented above requires full typing derivations for terms, something that is unrealistic to keep around as part of the runtime representation of terms. Here we will present an informal refinement of the above algorithm, that works on suitably annotated terms, instead of full typing derivations. The annotations are the minimal possible extra information needed to simulate the above algorithm.

Definition D.36 *We define a notion of annotated terms, for which we reuse the t syntactic class; it will be apparent from the context whether we mean a normal or an annotated term.*

$$t ::= c \mid s \mid f_{\mathbf{I}} \mid b_i \mid \lambda(t_1).t_2 \mid \Pi_s(t_1).t_2 \mid (t_1 : t)\, t_2 \mid t_1 =_t t_2 \mid X_i/\sigma$$

Lemma D.37 *1. If t is an unannotated term with $\bullet$, Ψ_u; $\Phi \vdash_p t : t'$ then there exists a derivation for Ψ_u; $\Phi \vdash_p t : t'$ where all terms are annotated terms.*

2. The inverse is also true.

These are trivial to prove by structural induction on the typing derivations.

Definition D.38 *The unification procedure is defined through the following judgement. It gets Ψ as a global parameter, which we omit here.*

$\boxed{(T) \sim (T')}$

$$\frac{(t) \sim \left(t'\right) \triangleleft \mathsf{unspec}_\Psi \triangleright \widehat{\sigma_\Psi}}{([\Phi]\, t) \sim \left([\Phi]\, t'\right) \triangleright \widehat{\sigma_\Psi}} \qquad \frac{\left(\Phi,\, \Phi'\right) \sim \left(\Phi,\, \Phi''\right) \triangleright \widehat{\sigma_\Psi}}{\left([\Phi]\, \Phi'\right) \sim \left([\Phi]\, \Phi''\right) \triangleright \widehat{\sigma_\Psi}}$$

$\boxed{(\Phi') \sim (\Phi'')}$

$$\frac{}{(\bullet) \sim (\bullet) \triangleright \mathsf{unspec}_{\widehat{\Psi}}} \qquad \frac{(\Phi) \sim \left(\Phi'\right) \triangleright \widehat{\sigma_\Psi} \qquad (t) \sim \left(t'\right) \triangleleft \widehat{\sigma_\Psi}\widehat{\sigma_\Psi}'}{((\Phi,\, t)) \sim \left((\Phi',\, t')\right) \triangleright \widehat{\sigma_\Psi}'}$$

$$\frac{}{(\Phi,\, X_i) \sim \left(\Phi, \Phi'\right) = \mathsf{unspec}_{\widehat{\Psi}}[i \mapsto [\Phi]\Phi']}$$

$\boxed{(t) \sim (t')}$

$$\frac{}{(c) \sim (c) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}} \qquad \frac{}{(s) \sim (s) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}} \qquad \frac{\mathbf{I} \cdot \widehat{\sigma_\Psi} = \mathbf{I}'}{(f_{\mathbf{I}}) \sim \left(f'_{\mathbf{I}}\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}}$$

$$\frac{s = s' \qquad (t_1) \sim \left(t'_1\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}' \qquad (\lceil t_2 \rceil) \sim \left(\lceil t'_2 \rceil\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_\Psi}''}{(\Pi_s(t_1).t_2) \sim \left(\Pi_{s'}(t'_1).t'_2\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}''} \qquad \frac{(\lceil t_2 \rceil) \sim \left(\lceil t'_2 \rceil\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}'}{(\lambda(t_1).t_2) \sim \left(\lambda(t'_1).t'_2\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}'}$$

$$\frac{(t) \sim \left(t'\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}' \qquad (t_1) \sim \left(t'_1\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_{\Psi 1}} \qquad (t_2) \sim \left(t'_2\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_{\Psi 2}} \qquad \widehat{\sigma_{\Psi 1}} \circ \widehat{\sigma_{\Psi 2}} = \widehat{\sigma_\Psi}''}{((t_1 : t)\ t_2) \sim \left(\left(t'_1 : t'\right)\ t'_2\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}''}$$

$$\frac{(t) \sim \left(t'\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}' \qquad (t_1) \sim \left(t'_1\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_{\Psi 1}} \qquad (t_2) \sim \left(t'_2\right) \lhd \widehat{\sigma_\Psi}' \rhd \widehat{\sigma_{\Psi 2}} \qquad \widehat{\sigma_{\Psi 1}} \circ \widehat{\sigma_{\Psi 2}} = \widehat{\sigma_\Psi}''}{(t_1 =_t t_2) \sim \left(t'_1 =_{t'} t'_2\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}''}$$

$$\frac{\widehat{\sigma_\Psi}.i = ? \qquad t' <^f |\sigma \cdot \widehat{\sigma_\Psi}|}{(X_i/\sigma) \sim \left(t'\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}[i \mapsto t']} \qquad \frac{\widehat{\sigma_\Psi}.i = t'}{(X_i/\sigma) \sim \left(t'\right) \lhd \widehat{\sigma_\Psi} \rhd \widehat{\sigma_\Psi}}$$

E. Simple staging support

Here we will add a light-weight staging support to the computational language. We extend the computational language as follows.

Definition E.1 *The syntax of the computational language is extended below.*

$$\begin{array}{l} e ::= \cdots \mid \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' \\ \Gamma ::= \cdots \mid \Gamma,\ x :_s \tau \end{array}$$

Definition E.2 *Freshening and binding for computational types and terms are extended as follows.*

$\boxed{\lceil e \rceil^M_{N,K}}$

$$\lceil \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' \rceil^M_{N,K} \quad = \quad \mathsf{letstatic}\ x\ =\ \lceil e \rceil^M_{N,K}\ \mathsf{in}\ \lceil e' \rceil^{M+1}_{N,K}$$

$\boxed{\lfloor e \rfloor^M_{N,K}}$

$$\lfloor \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' \rfloor^M_{N,K} \quad = \quad \mathsf{letstatic}\ x\ =\ \lfloor e \rfloor^M_{N,K}\ \mathsf{in}\ \lfloor e' \rfloor^{M+1}_{N,K}$$

Definition E.3 *Extension substitution application to computational-level types and terms.*

$\boxed{e \cdot \sigma_\Psi}$

$$\mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' \cdot \sigma_\Psi \quad = \quad \mathsf{letstatic}\ x\ =\ e \cdot \sigma_\Psi\ \mathsf{in}\ e' \cdot \sigma_\Psi$$

Definition E.4 *Limiting a context to the static types is defined as follows.*

$\boxed{\Gamma|_{\text{static}}}$

$$\begin{array}{lcl} \bullet|_{\text{static}} & = & \bullet \\ (\Gamma,\, x :_s t)|_{\text{static}} & = & \Gamma|_{\text{static}},\, x : t \\ (\Gamma,\, x : t)|_{\text{static}} & = & \Gamma|_{\text{static}} \\ (\Gamma,\, \alpha : k)|_{\text{static}} & = & \Gamma|_{\text{static}} \end{array}$$

Definition E.5 *The typing judgements for the computational language are extended below.*

$\boxed{\Psi;\, \Sigma;\, \Gamma \vdash e : \tau}$

$$\frac{\bullet;\, \Sigma;\, \Gamma|_{\text{static}} \vdash e : \tau \qquad \Psi;\, \Sigma;\, \Gamma, x :_s \tau \vdash e' : \tau}{\Psi;\, \Sigma;\, \Gamma \vdash \mathsf{letstatic}\, x \,=\, e\, \mathsf{in}\, e' : \tau} \qquad \frac{x :_s \tau \in \Gamma}{\Psi;\, \Sigma;\, \Gamma \vdash x : \tau}$$

Definition E.6 *Small-step operational semantics for the language are extended below.*

$$\begin{array}{rcl} e & ::= & \Lambda(K).e \mid e\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ e \mid \mathsf{unpack}\ e\ (.)x.(e') \\ & & \mid () \mid \mathsf{error} \mid \lambda x : \tau.e \mid e\ e' \mid x \mid (e,\, e') \mid \mathsf{proj}_i\ e \mid \mathsf{inj}_i\ e \mid \mathsf{case}(e, x.e', x.e'') \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \mid \mathsf{ref}\ e \\ & & \mid e := e' \mid !e \mid l \mid \Lambda\alpha : k.e \mid e\ \tau \mid \mathsf{fix}\ x : \tau.e \\ & & \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e') \mid \mathsf{letstatic}\, x \,=\, e\, \mathsf{in}\, e' \\ v & ::= & \Lambda(K).e_d \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ v \mid () \mid \lambda x : \tau.e_d \mid (v,\, v') \mid \mathsf{inj}_i\ v \mid \mathsf{fold}\ v \mid l \mid \Lambda\alpha : k.e_d \\ e_d & ::= & \Lambda(K).e_d \mid e_d\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ e_d \mid \mathsf{unpack}\ e_d\ (.)x.(e'_d) \\ & & \mid () \mid \mathsf{error} \mid \lambda x : \tau.e_d \mid e_d\ e'_d \mid x \mid (e_d,\, e'_d) \mid \mathsf{proj}_i\ e_d \mid \mathsf{inj}_i\ e_d \mid \mathsf{case}(e_d, x.e'_d, x.e''_d) \mid \mathsf{fold}\ e_d \mid \mathsf{unfold}\ e_d \\ & & \mid \mathsf{ref}\ e_d \mid e_d := e'_d \mid !e_d \mid l \mid \Lambda\alpha : k.e_d \mid e_d\ \tau \mid \mathsf{fix}\ x : \tau.e_d \\ & & \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto e'_d) \\ \mathcal{S} & ::= & \mathsf{letstatic}\, x \,=\, \bullet\, \mathsf{in}\, e' \mid \mathsf{letstatic}\, x \,=\, \mathcal{S}\, \mathsf{in}\, e' \mid \Lambda(K).\mathcal{S} \mid \lambda x : \tau.\mathcal{S} \mid \mathsf{unpack}\ e_d\ (.)x.(\mathcal{S}) \\ & & \mid \mathsf{case}(e_d, x.\mathcal{S}, x.e_2) \mid \mathsf{case}(e_d, x.e_d, x.\mathcal{S}) \mid \Lambda\alpha : k.\mathcal{S} \mid \mathsf{fix}\ x : \tau.\mathcal{S} \mid \mathsf{unify}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ (\Psi.T' \mapsto \mathcal{S}) \\ & & \mid \mathcal{E}_s[\mathcal{S}] \\ \mathcal{E}_s & ::= & \mathcal{E}_s\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ \mathcal{E}_s \mid \mathsf{unpack}\ \mathcal{E}_s\ (.)x.(e') \mid \mathcal{E}_s\ e' \mid e_d\ \mathcal{E}_s \mid (\mathcal{E}_s,\, e) \mid (e_d,\, \mathcal{E}_s) \mid \mathsf{proj}_i\ \mathcal{E}_s \\ & & \mid \mathsf{inj}_i\ \mathcal{E}_s \mid \mathsf{case}(\mathcal{E}_s, x.e_1, x.e_2) \mid \mathsf{fold}\ \mathcal{E}_s \mid \mathsf{unfold}\ \mathcal{E}_s \mid \mathsf{ref}\ \mathcal{E}_s \mid \mathcal{E}_s := e' \mid e_d := \mathcal{E}_s \mid !\mathcal{E}_s \mid \mathcal{E}_s\ \tau \\ \mathcal{E} & ::= & \bullet \mid \mathcal{E}\ T \mid \mathsf{pack}\ T\ \mathsf{return}\ (.\tau)\ \mathsf{with}\ \mathcal{E} \mid \mathsf{unpack}\ \mathcal{E}\ (.)x.(e_d) \mid \mathcal{E}\ e_d \mid v\ \mathcal{E} \mid (\mathcal{E},\, e_d) \mid (v,\, \mathcal{E}) \mid \mathsf{proj}_i\ \mathcal{E} \mid \mathsf{inj}_i\ \mathcal{E} \\ & & \mid \mathsf{case}(\mathcal{E}, x.e'_d, x.e''_d) \mid \mathsf{fold}\ \mathcal{E} \mid \mathsf{unfold}\ \mathcal{E} \mid \mathsf{ref}\ \mathcal{E} \mid \mathcal{E} := e_d \mid v := \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E}\ \tau \\ \mu & ::= & \bullet \mid \mu,\, l \mapsto v \end{array}$$

$\boxed{(\,\mu\,,\, e\,) \longrightarrow_s ((\,\mu\,,\, e'\,)|\mathsf{error})}$

$$\frac{(\,\mu\,,\, e_d\,) \longrightarrow (\,\mu'\,,\, e'_d\,)}{(\,\mu\,,\, \mathcal{S}[e_d]\,) \longrightarrow_s (\,\mu'\,,\, \mathcal{S}[e'_d]\,)} \qquad \frac{(\,\mu\,,\, e_d\,) \longrightarrow \mathsf{error}}{(\,\mu\,,\, \mathcal{S}[e_d]\,) \longrightarrow_s \mathsf{error}} \qquad (\,\mu\,,\, \mathcal{S}[\mathsf{letstatic}\, x \,=\, v\, \mathsf{in}\, e]\,) \longrightarrow_s (\,\mu\,,\, \mathcal{S}[e[v/x]]\,)$$

$$(\,\mu\,,\, \mathsf{letstatic}\, x \,=\, v\, \mathsf{in}\, e\,) \longrightarrow_s (\,\mu\,,\, e[v/x]\,)$$

Most lemmas are trivial to adapt. We adapt the substitution lemma for computational terms below.

Lemma E.7 (Substitution) *1. If* $\Psi,\, \Psi';\, \Gamma,\, \alpha' : k',\, \Gamma' \vdash \tau : k$ *and* $\Psi;\, \Gamma \vdash \tau' : k'$ *then* $\Psi,\, \Psi';\, \Gamma,\, \Gamma'[\tau'/\alpha'] \vdash \tau[\tau'/\alpha'] : k$.

2. *If* $\Psi,\ \Psi';\ \Sigma\ \Gamma,\ \alpha' : k',\ \Gamma' \vdash e : \tau$ *and* $\Psi;\ \Gamma \vdash \tau' : k'$ *then* $\Psi,\ \Psi';\ \Sigma;\ \Gamma,\ \Gamma'[\tau'/\alpha'] \vdash e[\tau'/\alpha'] : \tau[\tau'/\alpha']$.

3. *If* $\Psi,\ \Psi';\ \Sigma\ \Gamma,\ x' : \tau',\ \Gamma' \vdash e_d : \tau$ *and* $\Psi;\ \Sigma;\ \Gamma \vdash e'_d : \tau'$ *then* $\Psi,\ \Psi';\ \Sigma;\ \Gamma,\ \Gamma' \vdash e_d[e'_d/x'] : \tau$.

4. *If* $\Psi;\ \Gamma,\ x :_s \tau,\ \Gamma' \vdash e : \tau'$ *and* $\bullet;\ \Sigma;\ \bullet \vdash v : \tau$ *then* $\Psi;\ \Sigma;\ \Gamma,\ \Gamma' \vdash e[v/x] : \tau'$.

5. *If* $\Psi;\ \Gamma,\ x : \tau,\ \Gamma' \vdash e : \tau'$ *and* $\bullet;\ \Sigma;\ \bullet \vdash v : \tau$ *then* $\Psi;\ \Sigma;\ \Gamma,\ \Gamma' \vdash e[v/x] : \tau'$.

Easy proof by structural induction on the typing derivation for e. We prove the interesting cases below:

Part 3. **Case** $\dfrac{x :_s \tau \in \Gamma}{\Psi;\ \Sigma;\ \Gamma \vdash x : \tau}\ \triangleright$

We have that $e_d[e'_d/x] = e'_d$, and $\Psi;\ \Sigma;\ \Gamma \vdash e'_d : \tau$, which is the desired.

Case $\dfrac{\bullet;\ \Sigma;\ \Gamma|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \Sigma;\ \Gamma,\ x :_s \tau \vdash e' : \tau'}{\Psi;\ \Sigma;\ \Gamma \vdash \mathsf{letstatic}\ x\ =\ e\ \mathsf{in}\ e' : \tau'}\ \triangleright$

Impossible case, because the theorem only has to do with e_d cases.

Part 4. Most cases are trivial. The only interesting case follows.

Case $\dfrac{\bullet;\ \Sigma;\ \Gamma|_{\text{static}},\ x : \tau,\ \Gamma'|_{\text{static}} \vdash e : \tau \qquad \Psi;\ \Sigma;\ \Gamma,\ x :_s \tau,\ \Gamma',\ x' :_s \tau'' \vdash e' : \tau'}{\Psi;\ \Sigma;\ \Gamma,\ x :_s \tau,\ \Gamma' \vdash \mathsf{letstatic}\ x'\ =\ e\ \mathsf{in}\ e' : \tau'}\ \triangleright$

We use part 5 for e to get that $\bullet;\ \Sigma;\ \Gamma|_{\text{static}},\ \Gamma'|_{\text{static}} \vdash e[v/x] : \tau$.
By induction hypothesis for e' we get $\Psi;\ \Sigma;\ \Gamma,\ \Gamma', x' :_s \tau'' \vdash e'[v/x] : \tau'$.
Thus using the same typing rule we get the desired result.

Part 5. Trivial by structural induction.

Lemma E.8 (Types of decompositions) *1. If* $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{S}[e] : \tau$ *with* $\Gamma|_{static} = \bullet$*, then there exists* τ' *such that* $\bullet;\ \Sigma;\ \bullet \vdash e : \tau'$ *and for all* e' *such that* $\bullet;\ \Sigma;\ \bullet \vdash e' : \tau'$*, we have that* $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{S}[e'] : \tau$.

2. *If* $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{E}_s[e] : \tau$ *then there exists* τ' *such that* $\Psi;\ \Sigma;\ \Gamma \vdash e : \tau'$ *and for all* e' *such that* $\Psi;\ \Sigma;\ \Gamma \vdash e' : \tau'$*, we have that* $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{E}_s[e'] : \tau$.

Part 1. By structural induction on $\mathcal{S}$.

Case $\mathcal{S} = \mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e'\ \triangleright$ By inversion of typing we get that $\bullet;\ \Sigma;\ \Gamma|_{\text{static}} \vdash e : \tau$. We have $\Gamma|_{\text{static}}$, thus we get $\bullet;\ \Sigma;\ \bullet \vdash e : \tau'$. Using the same typing rule we get the desired result for $\mathcal{S}[e']$.

Case $\mathcal{S} = \mathsf{letstatic}\ x\ =\ \mathcal{S}'\ \mathsf{in}\ e''\ \triangleright$ By inductive hypothesis for $\mathcal{S}'$ we get the desired directly.

Case $\mathcal{S} = \Lambda(K).\mathcal{S}'\ \triangleright$ We have that $\lceil \mathcal{S}'[e_d] \rceil = \mathcal{S}''[\lceil e \rceil]$ with $\mathcal{S}'' = \lceil \mathcal{S}'[\bullet] \rceil$. By inductive hypothesis for $\Psi,\ K;\ \Sigma;\ \Gamma \vdash \mathcal{S}''[\lceil e \rceil] : \tau$ we get that $\bullet;\ \Sigma;\ \bullet \vdash \lceil e \rceil : \tau'$. From this we directly get $\lceil e \rceil = e$, and the desired follows immediately (using the rest of the inductive hypothesis).

Case $\mathcal{S} = \mathcal{E}_s[\mathcal{S}]\ \triangleright$ We have that $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{E}_s[\mathcal{S}[e_d]] : \tau$. Using part 2 for $\mathcal{E}_s$ and $\mathcal{S}[e_d]$ we get that $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{S}[e_d] : \tau'$ for some τ' and also that for all e' such that $\Psi;\ \Sigma;\ \Gamma \vdash e : \tau'$, $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{E}_s[e'] : \tau$. Then using induction hypothesis we get a τ'' such that $\bullet;\ \Sigma;\ \bullet \vdash e_d : \tau''$. For this type, we also have that $\bullet;\ \Sigma;\ \bullet \vdash e'_d : \tau''$ implies $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{S}[e'_d] : \tau'$, which further implies $\Psi;\ \Sigma;\ \Gamma \vdash \mathcal{E}_s[\mathcal{S}[e'_d]] : \tau$.

The rest of the cases follow similar ideas.

Part 2. By induction on the structure of $\mathcal{E}_s$. In each case, use inversion of typing to get the type for e, and then use the same typing rule to get the derivation for $\mathcal{E}_s[e']$.

Theorem E.9 (Preservation) *1. If* $\bullet; \Sigma; \bullet \vdash e : \tau$, $\mu \sim \Sigma$, $(\mu, e) \longrightarrow_s (\mu', e')$ *then there exists* Σ' *such that* $\Sigma \subseteq \Sigma'$, $\mu' \sim \Sigma'$ *and* $\bullet; \Sigma'; \bullet \vdash e' : \tau$.

2. If $\bullet; \Sigma; \bullet \vdash e_d : \tau$, $\mu \sim \Sigma$, $(\mu, e_d) \longrightarrow_s (\mu', e'_d)$ *then there exists* Σ' *such that* $\Sigma \subseteq \Sigma'$, $\mu' \sim \Sigma'$ *and* $\bullet; \Sigma'; \bullet \vdash e'_d : \tau$.

Part 1 We proceed by induction on the derivation of $(\mu, e) \longrightarrow (\mu', e')$.

Case $\dfrac{(\mu, e_d) \longrightarrow (\mu', e'_d)}{(\mu, \mathcal{S}[e_d]) \longrightarrow_s (\mu', \mathcal{S}[e'_d])}$ $\triangleright$
Using the lemma E.8 we get $\bullet; \Sigma; \bullet \vdash e_d : \tau'$. Using part 2, we get that $\bullet; \Sigma; \bullet \vdash e'_d : \tau'$. Thus, using again the same lemma we get the desired.

Case $(\mu, \mathcal{S}[\mathsf{letstatic}\, x = v \,\mathsf{in}\, e]) \longrightarrow_s (\mu, \mathcal{S}[e[v/x]])$ $\triangleright$
Using the lemma E.8 we get $\bullet; \Sigma; \bullet \vdash \mathsf{letstatic}\, x = v \,\mathsf{in}\, e : \tau'$. By typing inversion we get that $\bullet; \Sigma; \bullet \vdash v : \tau''$, and also that $\bullet; \Sigma; x :_s \tau'' \vdash e : \tau'$. Using the substitution lemma E.7 we get the desired result.

The rest of the cases are trivial.

Part 2 Proceeds exactly as before, as e_d entirely matches the definition of expressions prior to the extension.

Theorem E.10 (Unique decomposition) *1. For every expression e, we have:*

(a) Either e is a dynamic expression e_d*, in which case there is no way to write* e_d *as* $\mathcal{S}[e']$ *for any* e'.

(b) Or there is a unique decomposition of e into $\mathcal{S}[e_d]$.

2. For every expression e_d*, we have:*

(a) Either it is a value v and the decomposition $v = \mathcal{E}[e]$ *implies* $\mathcal{E} = \bullet$ *and* $e = v$.

(b) Or, there is a unique decomposition of e_d *into* $e_d = \mathcal{E}[v]$.

Part 1. Proceed by induction on the structure of the expression e.

Case $\Lambda(K).e'$ $\triangleright$ By induction hypothesis on the structure of e'. If we have $e' = e_d$, then this is a dynamic expression already. In the other cases, we get a unique decomposition of e' into $\mathcal{S}'[e'']$. The original expression e can be uniquely decomposed using $\mathcal{S} = \Lambda(K).\mathcal{S}'$, with $e = \mathcal{S}[e'']$. This decomposition is unique because the outer frame is uniquely determined; if the inner frames or the expression filling the hole could be different, we would violate the uniqueness part of the decomposition returned by induction hypothesis.

Case $e'\ T$ $\triangleright$ By induction hypothesis we get that either $e' = e'_d$, or there is a unique decomposition of e' into $\mathcal{S}'[e'']$. In that case, e is uniquely decomposed using $\mathcal{S} = \mathcal{E}_s[\mathcal{S}']$ with $\mathcal{E}_s = \bullet\ T$, into $e = \mathcal{S}'[e'']\ T$.

Case $\mathsf{unpack}\, x\, (.)e'.(e'')$ $\triangleright$ By induction hypothesis on e'; if it is a dynamic expression, then by induction hypothesis on e''; if that too is a dynamic expression, then the original expression is too. Otherwise, use the unique decomposition of $e'' = \mathcal{S}'[e''']$ to get the unique decomposition $e = \mathsf{unpack}\, x\, (.)e_d.(\mathcal{S}'[e'''])$. If e' is not a dynamic expression, use the unique decomposition of $e' = \mathcal{S}''[e'''']$ to get the unique decomposition $e = \mathsf{unpack}\, x\, (.)\mathcal{S}''[e''''].(e'')$.

Case $\mathsf{letstatic}\ x\ =\ e'\ \mathsf{in}\ e''\ \triangleright$ By induction hypothesis on e'.
In the case that $e' = e_d$, then trivially we have the unique decomposition $e = (\mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e'')[e_d]$.
In the case where $e' = \mathcal{S}[e_d]$, we have the unique decomposition $e = (\mathsf{letstatic}\ x\ =\ \mathcal{S}\ \mathsf{in}\ e'')[e_d]$.

The rest of the cases are similar.

Part 2. Trivial by induction on the structure of the dynamic expression e_d.

Theorem E.11 (Progress) *1. If $\bullet; \Sigma; \bullet \vdash e : \tau$ and $\mu \sim \Sigma$, then either $\mu,\ e \longrightarrow_s \mathsf{error}$, or e is a dynamic expression e_d, or there exist μ' and e' such that $\mu,\ e \longrightarrow_s \mu',\ e'$.*

2. If $\bullet; \Sigma; \bullet \vdash e_d : \tau$ and $\mu \sim \Sigma$, then either $\mu,\ e_d \longrightarrow \mathsf{error}$, or e_d is a value v, or there exist μ' and e'_d such that $\mu,\ e_d \longrightarrow \mu',\ e'_d$.

Part 1 First, we use the unique decomposition lemma E.10, we get that either e is a dynamic expression, in which case we are done; or a decomposition into $\mathcal{S}[e'_d]$. In that case, we use the lemma E.8 and part 2 to get that either e'_d is a value or that some progress can be made: either by failing or getting a μ', e''_d, in which case we use the appropriate rule for $\longrightarrow_s$ either to fail or to progress to $\mu', \mathcal{S}[e''_d]$. If e'_d is a value, then we split cases depending on $\mathcal{S}$ – if it is simply $\mathsf{letstatic}\ x\ =\ \bullet\ \mathsf{in}\ e$ or it is nested. In both cases we make progress using the appropriate step rule.

Part 2 Identical as before.

F. Collapsing terms with extension variables into terms with normal variables

Definition F.1 *A decidable judgement for deciding whether a term t, a context Φ, etc. are collapsable to a normal logical term is given below.*

Intuitively, it defines as collapsable terms where all context Φ involved (even inside extension variable types) are subcontexts of a single context Φ'' (which is the result of the procedure), and all extension variables are used with identity substitutions of that context.

$\boxed{\mathsf{collapsible}\,(\Psi) \triangleleft \Phi' \triangleright \Phi''}$

$$\frac{}{\mathsf{collapsible}\,(\bullet) \triangleleft \Phi' \triangleright \Phi'} \qquad \frac{\mathsf{collapsible}\,(\Psi) \triangleleft \Phi' \triangleright \Phi'' \qquad \mathsf{collapsible}\,(K) \triangleleft \Phi'' \triangleright \Phi'''}{\mathsf{collapsible}\,(\Psi,\ K) \triangleleft \Phi' \triangleright \Phi'''}$$

$\boxed{\mathsf{collapsible}\,(K) \triangleleft \Phi' \triangleright \Phi''}$

$$\frac{K = T \qquad \mathsf{collapsible}\,(T) \triangleleft \Phi' \triangleright \Phi''}{\mathsf{collapsible}\,(K) \triangleleft \Phi' \triangleright \Phi''} \qquad \frac{\mathsf{collapsible}\,(\Phi) \triangleleft \Phi' \triangleright \Phi''}{\mathsf{collapsible}\,([\Phi]\,\mathsf{ctx}) \triangleleft \Phi' \triangleright \Phi''}$$

$\boxed{\mathsf{collapsible}\,(T) \triangleleft \Phi' \triangleright \Phi''}$

$$\frac{\mathsf{collapsible}\,(\Phi) \triangleleft \Phi' \triangleright \Phi'' \qquad \mathsf{collapsible}\,(t) \triangleleft \Phi''}{\mathsf{collapsible}\,([\Phi]\,t) \triangleleft \Phi' \triangleright \Phi''} \qquad \frac{\mathsf{collapsible}\,(\Phi_1,\ \Phi_2) \triangleleft \Phi' \triangleright \Phi''}{\mathsf{collapsible}\,([\Phi_1]\Phi_2) \triangleleft \Phi' \triangleright \Phi''}$$

$\boxed{\text{collapsible}\,(\Phi) \lhd \Phi' \rhd \Phi''}$

$$\frac{}{\text{collapsible}\,(\bullet) \lhd \Phi' \rhd \Phi'} \qquad \frac{\text{collapsible}\,(\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi = \Phi'' \qquad \text{collapsible}\,(t) \lhd \Phi}{\text{collapsible}\,(\Phi,\, t) \lhd \Phi' \rhd (\Phi,\, t)}$$

$$\frac{\text{collapsible}\,(\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi \subset \Phi'' \qquad \text{collapsible}\,(t) \lhd \Phi''}{\text{collapsible}\,(\Phi,\, t) \lhd \Phi' \rhd \Phi''} \qquad \frac{\text{collapsible}\,(\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi = \Phi''}{\text{collapsible}\,(\Phi,\, X_i) \lhd \Phi' \rhd (\Phi,\, X_i)}$$

$$\frac{\text{collapsible}\,(\Phi) \lhd \Phi' \rhd \Phi'' \qquad \Phi \subset \Phi''}{\text{collapsible}\,(\Phi,\, X_i) \lhd \Phi' \rhd \Phi''}$$

$\boxed{\text{collapsible}\,(t) \lhd \Phi'}$

$$\frac{}{\text{collapsible}\,(s) \lhd \Phi'} \qquad \frac{}{\text{collapsible}\,(c) \lhd \Phi'} \qquad \frac{}{\text{collapsible}\,(f_{\mathbf{I}}) \lhd \Phi'}$$

$$\frac{\text{collapsible}\,(t_1) \lhd \Phi' \qquad \text{collapsible}\,(\lceil t_2 \rceil) \lhd \Phi'}{\text{collapsible}\,(\lambda(t_1).t_2) \lhd \Phi'} \qquad \frac{\text{collapsible}\,(t_1) \lhd \Phi' \qquad \text{collapsible}\,(\lceil t_2 \rceil) \lhd \Phi'}{\text{collapsible}\,(\Pi(t_1).t_2) \lhd \Phi'}$$

$$\frac{\text{collapsible}\,(t_1) \lhd \Phi' \qquad \text{collapsible}\,(t_2) \lhd \Phi'}{\text{collapsible}\,(t1\ t_2) \lhd \Phi'} \qquad \frac{\text{collapsible}\,(t_1) \lhd \Phi' \qquad \text{collapsible}\,(t_2) \lhd \Phi'}{\text{collapsible}\,(t1 = t_2) \lhd \Phi'}$$

$$\frac{\sigma \subseteq \text{id}\Phi'}{\text{collapsible}\,(X_i/\sigma) \lhd \Phi'}$$

$\boxed{\text{collapsible}\,(\Psi \vdash T : K) \rhd \Phi'}$

$$\frac{\text{collapsible}\,(\Psi) \lhd \bullet \rhd \Phi' \qquad \text{collapsible}\,(K) \lhd \Phi' \rhd \Phi'' \qquad \text{collapsible}\,(T) \lhd \Phi'' \rhd \Phi''}{\text{collapsible}\,(\Psi \vdash T : K) \rhd \Phi''}$$

$\boxed{\text{collapsible}\,(\Psi \vdash \Phi\ \text{wf}) \rhd \Phi'}$

$$\frac{\text{collapsible}\,(\Psi) \lhd \bullet \rhd \Phi' \qquad \text{collapsible}\,(\Phi) \lhd \Phi' \rhd \Phi''}{\text{collapsible}\,(\Psi \vdash \Phi\ \text{wf}) \rhd \Phi''}$$

Lemma F.2 *1. If collapsible* $(\Phi) \lhd \Phi' \rhd \Phi''$ *then either* $\Phi' \subseteq \Phi$ *and* $\Phi'' = \Phi$*, or* $\Phi \subseteq \Phi'$ *and* $\Phi'' = \Phi'$*.*

2. If collapsible $(\Psi \vdash [\Phi]t : [\Phi]t_T) \rhd \Phi'$ *then* $\Phi \subseteq \Phi'$*.*

3. If collapsible $(\Psi \vdash [\Phi_0]\Phi_1 : [\Phi_0]\Phi_1) \rhd \Phi'$ *then* $\Phi_0, \Phi_1 \subseteq \Phi'$*.*

Trivial by structural induction.

Lemma F.3 *1. If collapsible* $(\Psi) \lhd \bullet \rhd \Phi$ *and* $\Phi \subseteq \Phi'$ *then collapsible* $(\Psi) \lhd \Phi' \rhd \Phi'$*.*

2. If collapsible $(K) \lhd \bullet \rhd \Phi$ *and* $\Phi \subseteq \Phi'$ *then collapsible* $(K) \lhd \Phi' \rhd \Phi'$*.*

3. If collapsible $(T) \lhd \bullet \rhd \Phi$ *and* $\Phi \subseteq \Phi'$ *then collapsible* $(T) \lhd \Phi' \rhd \Phi'$*.*

4. If collapsible $(\Phi_0) \lhd \bullet \rhd \Phi$ *and* $\Phi \subseteq \Phi'$ *then collapsible* $(\Phi_0) \lhd \Phi' \rhd \Phi'$*.*

Trivial by structural induction on the collapsing relation.

Lemma F.4 *If* $\vdash \Psi$ *wf and* $\mathsf{collapsible}\,(\Psi) \lhd \bullet \rhd \Phi^0$*, then there exist* Ψ^1*,* σ^1_Ψ*,* σ^2_Ψ*,* Φ^1*,* σ^1 *and* σ^{-1} *such that:*
$\Psi^1 \vdash \sigma^1_\Psi : \Psi$*,*
$\bullet \vdash \sigma^2_\Psi : \Psi^1$*,*
$\Psi^1 \vdash \Phi^1$ *wf,*
$\Psi^1;\, \Phi^1 \vdash \sigma^1 : \Phi^0 \cdot \sigma^1_\Psi$*,*
$\Psi;\, \Phi^0 \vdash \sigma^{-1} : \Phi^1 \cdot \sigma^2_\Psi$*,*
for all t *such that* $\Psi;\, \Phi^0 \vdash t : t'$*, we have* $t \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{-1} = t$*, and all members of* Ψ^1 *are of the form* $[\Phi^*]t$ *where* $\Phi^* \subseteq \Phi^1$*.*

By induction on the derivation of the relation $\mathsf{collapsible}\,(\Psi) \lhd \bullet \rhd \Phi^0$.

Case $\Psi = \bullet\ \rhd$
We choose $\Psi^1 = \bullet$; $\sigma^1_\Psi = \sigma^2_\Psi = \bullet$; $\Phi^1 = \bullet$; $\sigma^1 = \bullet$; $\sigma^{-1} = \bullet$ $\Phi^1 = \bullet$ and the desired trivially hold.

Case $\Psi = \Psi',\ [\Phi]\,\mathsf{ctx}\ \rhd$
From the collapsable relation, we get: $\mathsf{collapsible}\,(\Psi') \lhd \bullet \rhd \Phi'^0$, $\mathsf{collapsible}\,([\Phi]\,\mathsf{ctx}) \lhd \Phi'^0 \rhd \Phi^0$. By induction hypothesis for Ψ', get:
$\Psi'^1 \vdash \sigma'^1_\Psi : \Psi'$,
$\bullet \vdash \sigma'^2_\Psi : \Psi'^1$,
$\Psi'^1 \vdash \Phi'^1$ wf,
$\Psi'^1;\, \Phi'^1 \vdash \sigma'^1 : \Phi'^0 \cdot \sigma'^1_\Psi$,
$\Psi';\, \Phi'^0 \vdash \sigma'^{-1} : \Phi'^1 \cdot \sigma'^2_\Psi$,
for all t such that $\Psi';\, \Phi'^0 \vdash t : t'$, we have $t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma'^{-1} = t$, and all members of Ψ'^1 are of the form $[\Phi^*]t$ where $\Phi^* \subseteq \Phi'^1$.
By inversion of typing for $[\Phi]\,\mathsf{ctx}$ we get that $\Psi' \vdash \Phi$ wf.
We fix $\sigma^1_\Psi = \sigma'^1_\Psi,\ [\Phi'^0 \cdot \sigma'^1_\Psi]$ which is a valid choice as long as we select Ψ^1 so that $\Psi'^1 \subseteq \Psi^1$. This substitution has correct type by taking into account the substitution lemma for Φ'^0 and σ'^1_Ψ.
For choosing the rest, we proceed by induction on the derivation of $\Phi'^0 \subseteq \Phi^0$.
If $\Phi^0 = \Phi'^0$, then:
We have $\Phi \subseteq \Phi'^0$ because of the previous lemma.
Choose $\Psi^1 = \Psi'^1$; $\sigma^2_\Psi = \sigma'^2_\Psi$; $\Phi^1 = \Phi'^1$; $\sigma^1 = \sigma'^1$; $\sigma^{-1} = \sigma'^{-1}$.
Everything holds trivially, other than σ^1_Ψ typing. This too is easy to prove by taking into account the substitution lemma for Φ and σ'^1_Ψ. Also, σ'^{-1} typing uses extension variable weakening. Last, for the cancellation part, terms that are typed under Ψ are also typed under Ψ' so this part is trivial too.
If $\Phi^0 = \Phi'^0,\ t$, then: (here we abuse things slightly – by identifying the context and substitutions from induction hypothesis with the ones we already have: their properties are the same for the new Φ'^0)
We have $\Phi = \Phi^0 = \Phi'^0,\ t$ because of the previous lemma (Φ^0 is not Φ'^0 thus $\Phi^0 = \Phi$).
First, choose $\Phi^1 = \Phi'^1,\ t \cdot \sigma'^1_\Psi \cdot \sigma'^1$. This is a valid choice, because $\Psi';\, \Phi'^0 \vdash t : s$; by applying σ'^1_Ψ we get $\Psi'^1;\, \Phi'^0 \cdot \sigma'^1_\Psi \vdash t \cdot \sigma'^1_\Psi : s$; by applying σ'^1 we get $\Psi'^1;\, \Phi'^1 \vdash t \cdot \sigma'^1_\Psi \cdot \sigma'^1 : s$.
Thus $\Psi'^1 \vdash \Phi'^1,\ t \cdot \sigma'^1_\Psi \cdot \sigma'^1$ wf (and the Ψ^1 we will choose is supercontext of Ψ'^1).
Now, choose $\Psi^1 = \Psi'^1,\ [\Phi_1]\, t \cdot \sigma'^1_\Psi \cdot \sigma'^1$. This is well-formed because of what we proved above about the substituted t, taking weakening into account. Also, the condition for the contexts in Ψ^1 being subcontexts of Φ^1 obviously holds.

Choose $\sigma^2_\Psi = \sigma'^2_\Psi,\ [\Phi_1] f_{|\Phi'^1|}$. We have $\bullet \vdash \sigma^2_\Psi : \Psi^1$ directly by our construction.
Choose $\sigma^1 = \sigma'^1,\ f_{|\Phi'^1|}$. We have that this latter term can be typed as $\Psi^1;\ \Phi^1 \vdash f_{|\Phi'^1|} : t \cdot \sigma'^1_\Psi \cdot \sigma'^1$, and thus we have $\Psi^1;\ \Phi^1 \vdash \sigma^1 : \Phi'^0 \cdot \sigma'^1_\Psi,\ t \cdot \sigma'^1_\Psi$.
Choose $\sigma^{-1} = \sigma'^{-1},\ f_{|\Phi'^0|}$, which is typed correctly since $t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma^{-1} = t$. Last, assume $\Psi;\ \Phi'^0,\ t \vdash t_* : t'_*$. We prove $t_* \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{-1} = t_*$.

> First, t_* is also typed under Ψ' because t_* cannot use the newly-introduced variable directly (even in the case where it would be part of Φ_0, there's still no extension variable that has $X_{|\Psi'|}$ in its context).
> Thus it suffices to prove $t_* \cdot \sigma'^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{-1} = t_*$.
> Then proceed by structural induction on t_*. The only interesting case occurs when $t_* = f_{|\Phi'^0|}$, in which case we have:
> $f_{|\Phi'^0|} \cdot \sigma'^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{-1} = f_{|\Phi'^0 \cdot \sigma'^1_\Psi|} \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{-1} = f_{|\Phi'^1|} \cdot \sigma^2_\Psi \cdot \sigma^{-1} = f_{|\Phi'^1 \cdot \sigma^2_\Psi|} \cdot \sigma^{-1} = f_{|\Phi'^0|}$

If $\Phi^0 = \Phi'^0,\ X_i$:
By well-formedness inversion we get that $\Psi.i = [\Phi_*]\,\mathsf{ctx}$, and by repeated inversions of the collapsable relation we get $\Phi_* \subseteq \Phi'^0$.
Choose $\Phi^1 = \Phi'^1\ ; \Psi^1 = \Psi'^1\ ; \sigma^2_\Psi = \sigma'^2_\Psi; \sigma^1 = \sigma'^1; \sigma^{-1} = \sigma'^{-1}$.
Most desiderata are trivial. For σ^1, note that $(\Phi'^1,\ X_i) \cdot \sigma'^1_\Psi = \Phi'^1 \cdot \sigma'^1_\Psi$ since by construction we have that σ'^1_Ψ always substitutes parametric contexts by the empty context.
For cancellation, we need to prove that for all t such that $\Psi;\ \Phi'^0,\ X_i \vdash t_* : t'_*$, we have $t_* \cdot \sigma^1_\Psi \cdot \sigma^1 \cdot \sigma^2_\Psi \cdot \sigma^{-1} = t_*$. This is proved directly by noticing that t_* is typed also under Ψ' (if X_i was the just-introduced variable, it wouldn't be able to refer to itself).

Case $\Psi = \Psi',\ [\Phi] t\ \triangleright$
From the collapsable relation, we get: $\mathsf{collapsible}\,(\Psi') \triangleleft \bullet \triangleright \Phi'^0$, $\mathsf{collapsible}\,(\Phi) \triangleleft \Phi'^0 \triangleright \Phi^0$, $\mathsf{collapsible}\,(t) \triangleleft \Phi^0$. By induction hypothesis for Ψ', get:
$\Psi'^1 \vdash \sigma'^1_\Psi : \Psi'$,
$\bullet \vdash \sigma'^2_\Psi : \Psi'^1$,
$\Psi'^1 \vdash \Phi'^1$ wf,
$\Psi'^1;\ \Phi'^1 \vdash \sigma'^1 : \Phi'^0 \cdot \sigma'^1_\Psi$,
$\Psi';\ \Phi'^0 \vdash \sigma'^{-1} : \Phi'^1 \cdot \sigma'^2_\Psi$,
for all t such that $\Psi';\ \Phi'^0 \vdash t : t'$, we have $t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma'^{-1} = t$, and all members of Ψ'^1 are of the form $[\Phi^*] t$ where $\Phi^* \subseteq \Phi'^1$.
Also from typing inversion we get: $\Psi' \vdash \Phi$ wf and $\Psi';\ \Phi \vdash t : s$.
We proceed similarly as in the previous case, by induction on $\Phi'^0 \subseteq \Phi^0$, in order to redefine $\Psi'^1, \sigma'^1_\Psi, \sigma'^2_\Psi, \Phi'^1, \sigma'^1, \sigma^{-1}$ with the properties:
$\Psi'^1 \vdash \sigma'^1_\Psi : \Psi'$,
$\bullet \vdash \sigma'^2_\Psi : \Psi'^1$,
$\Psi'^1 \vdash \Phi'^1$ wf,
$\Psi'^1;\ \Phi'^1 \vdash \sigma'^1 : \Phi^0 \cdot \sigma'^1_\Psi$,
$\Psi';\ \Phi^0 \vdash \sigma'^{-1} : \Phi'^1 \cdot \sigma'^2_\Psi$,
for all t such that $\Psi';\ \Phi^0 \vdash t : t'$, we have $t \cdot \sigma'^1_\Psi \cdot \sigma'^1 \cdot \sigma'^2_\Psi \cdot \sigma'^{-1} = t$, and all members of Ψ'^1 are of the form $[\Phi^*] t$ where $\Phi^* \subseteq \Phi'^1$.
Now we have $\Phi \subseteq \Phi^0$ thus $\Psi';\ \Phi^0 \vdash t : s$.
By applying σ'^1_Ψ and then σ'^1 to t we get $\Psi'^1;\ \Phi'^1 \vdash t \cdot \sigma'^1_\Psi \cdot \sigma'^1 : s$. We can now choose $\Phi^1 = \Phi'^1,\ t \cdot \sigma'^1_\Psi \cdot \sigma'^1$.
Choose $\Psi^1 = \Psi'^1,\ [\Phi^1] t \cdot \sigma'^1_\Psi \cdot \sigma'^1$. It is obviously well-formed.
Now, will choose σ^1_Ψ:

Need to choose t^1 such that $\Psi^1;\ \Phi\cdot\sigma'^1_\Psi \vdash t^1 : t\cdot\sigma'^1_\Psi$.
Assuming $t^1 = X_{|\Phi'^1|}/\sigma$, we need $t\cdot\sigma'^1_\Psi\cdot\sigma'^1\cdot\sigma = t\cdot\sigma'^1_\Psi$ and $\Psi^1;\ \Phi\cdot\sigma'^1_\Psi \vdash \sigma : \Phi^1$.
Thus, what we require is the inverse of σ^1. By construction, there exists such a σ, because σ^1 is just a variable renaming. (Note that this is different from σ^{-1}.)
Therefore, set $\sigma^1_\Psi = \sigma'^1_\Psi,\ [\Phi]X_{|\Phi'^1|}/\sigma$, which has the desirable properties.
Choose $\sigma^2_\Psi = \sigma'^2_\Psi,\ [\Phi^1]f_{|\Phi'^1|}$. We trivially have $\bullet \vdash \sigma^2_\Psi : \Psi^1$.
Choose $\sigma^1 = \sigma'^1$, with typing holding obviously.
Choose $\sigma^{-1} = \sigma'^{-1},\ X_{|\Psi'|}/\mathsf{id}\Phi$. Consider the cancellation fact; typing is then possible.
It remains to prove that for all t_* such that $\Psi', [\Phi]t;\ \Phi^0 \vdash t_* : t'_*$, we have $t\cdot\sigma^1_\Psi\cdot\sigma^1\cdot\sigma^2_\Psi\cdot\sigma^{-1} = t$.
This is done by structural induction on t_*, with the interesting case being $t_* = X_{|\Psi'|}/\sigma_*$. By inversion of collapsable relation, we get that $\sigma_* = \mathsf{id}\Phi$.
Thus $(X_{|\Psi'|}/\mathsf{id}\Phi)\cdot\sigma^1_\Psi\cdot\sigma^1\cdot\sigma^2_\Psi\cdot\sigma^{-1} = (X_{|\Phi'^1|}/\sigma)\cdot(\mathsf{id}\Phi\cdot\sigma^1_\Psi)\cdot\sigma^1\cdot\sigma^2_\Psi\cdot\sigma^{-1} = (X_{|\Phi'^1|}/\sigma)\cdot(\mathsf{id}\Phi\cdot\sigma^1_\Psi)\cdot\sigma^1\cdot\sigma^2_\Psi\cdot\sigma^{-1} = (X_{|\Phi'^1|}/\sigma)\cdot\sigma^1\cdot\sigma^2_\Psi\cdot\sigma^{-1} = (X_{|\Phi'^1|}/(\sigma\cdot\sigma^1))\cdot\sigma^2_\Psi\cdot\sigma^{-1} = (X_{|\Phi'^1|}/(\mathsf{id}\Phi^1))\cdot\sigma^2_\Psi\cdot\sigma^{-1} = (f_{|\Phi'^1|}\cdot(\mathsf{id}\Phi^1\cdot\sigma^2_\Psi))\cdot\sigma^{-1} = (f_{|\Phi'^1|}\cdot\mathsf{id}\Phi^1\cdot\sigma^2_\Psi)\cdot\sigma^{-1} = f_{|\Phi'^1|\cdot\sigma^2_\Psi|}\cdot\sigma^{-1} = X_{|\Psi'|}/\mathsf{id}\Phi$.

Theorem F.5 *If $\Psi \vdash [\Phi]t : [\Phi]t_T$ and $\mathsf{collapsible}(\Psi \vdash [\Phi]t : [\Phi]t_T) = \Phi_*$, then there exist Φ', t', t'_T and σ such that $\bullet \vdash \Phi'$ wf, $\bullet \vdash [\Phi']t' : [\Phi']t'_T$, $\Psi;\ \Phi \vdash \sigma : \Phi'$, $t'\cdot\sigma = t$ and $t'_T\cdot\sigma = t_T$.*

Easy to prove using above lemma. Set $\Phi' = \Phi^1\cdot\sigma^2_\Psi$, $t' = t\cdot\sigma^1_\Psi\cdot\sigma^1\cdot\sigma^2_\Psi$, $t'_T = t_T\cdot\sigma^1_\Psi\cdot\sigma^1\cdot\sigma^2_\Psi$, and also set $\sigma = \sigma^{-1}$.

A Case for Behavior-Preserving Actions in Separation Logic

David Costanzo and Zhong Shao

Yale University

Abstract. Separation Logic is a widely-used tool that allows for local reasoning about imperative programs with pointers. A straightforward definition of this "local reasoning" is that, whenever a program runs safely on some state, adding more state would have no effect on the program's behavior. However, for a mix of technical and historical reasons, local reasoning is defined in a more subtle way, allowing a program to lose some behaviors when extra state is added. In this paper, we propose strengthening local reasoning to match the straightforward definition mentioned above. We argue that such a strengthening does not have any negative effect on the usability of Separation Logic, and we present four examples that illustrate how this strengthening simplifies some of the metatheoretical reasoning regarding Separation Logic. In one example, our change even results in a more powerful metatheory.

1 Introduction

Separation Logic [8, 13] is widely used for verifying the correctness of C-like imperative programs [9] that manipulate mutable data structures. It supports *local reasoning* [15]: if we know a program's behavior on some heap, then we can automatically infer something about its behavior on any larger heap. The concept of local reasoning is embodied as a logical inference rule, known as the *frame rule*. The frame rule allows us to extend a specification of a program's execution on a small heap to a specification of execution on a larger heap.

For the purpose of making Separation Logic extensible, it is common practice to abstract over the primitive commands of the programming language being used. By "primitive commands" here, we mean commands that are not defined in terms of other commands. Typical examples of primitive commands include variable assignment $x := E$ and heap update $[E] := E'$. One example of a non-primitive command is $\texttt{while}\, B\, \texttt{do}\, C$.

When we abstract over primitive commands, we need to make sure that we still have a sound logic. Specifically, it is possible for the frame rule to become unsound for certain primitive commands. In order to guarantee that this does not happen, certain "healthiness" conditions are required of primitive commands. We refer to these conditions together as "locality," since they guarantee soundness of the frame rule, and the frame rule is the embodiment of local reasoning.

As one might expect, locality in Separation Logic is defined in such a way that it is *precisely* strong enough to guarantee soundness of the frame rule. In other

words, the frame rule is sound *if and only if* all primitive commands are local. In this paper, we consider a strengthening of locality. Clearly, any strengthening will still guarantee soundness of the frame rule. The tradeoff, then, is that the stronger we make locality, the fewer primitive commands there will be that satisfy locality. We claim that we can strengthen locality to the point where: (1) the usage of the logic is unaffected — specifically, we do not lose the ability to model any primitive commands that are normally modeled in Separation Logic; (2) our strong locality is precisely the property that one would intuitively expect it to be — that the behavior of a program is completely independent from any unused state; and (3) we significantly simplify various technical work in the literature relating to metatheoretical facts about Separation Logic. We refer to our stronger notion of locality as "behavior preservation," because the behavior of a program is preserved when moving from a small state to a larger one.

We justify statement (1) above, that the usage of the logic is unaffected, in Section 3 by demonstrating a version of Separation Logic using the same primitive commands as the standard one presented in [13], for which our strong locality holds. We show that, even though we need to alter the state model of standard Separation Logic, we do not need to change any of the inference rules. We justify the second statement, that our strong locality preserves program behavior, in Section 2. We will also show that the standard, weaker notion of locality is not behavior-preserving. We provide some justification of the third statement, that behavior preservation significantly simplifies Separation Logic metatheory, in Section 5 by considering four specific examples in detail. As a primer, we will say a little bit about each example here.

The first simplification that we show is in regard to *program footprints*, as defined and analyzed in [12]. Informally, a footprint of a program is a set of states such that, given the program's behavior on those states, it is possible to infer all of the program's behavior on all other states. Footprints are useful for giving complete specifications of programs in a concise way. Intuitively, locality should tell us that the set of *smallest safe states*, or states containing the minimal amount of resources required for the program to safely execute, should always be a footprint. However, this is not the case in standard Separation Logic. To quote the authors in [12], the intuition that the smallest safe states should form a footprint "fails due to the subtle nature of the locality condition." We show that in the context of behavior-preserving locality, the set of smallest safe states does indeed form a footprint.

The second simplification regards the theory of data refinement, as defined in [6]. Data refinement is a formalism of the common programming paradigm in which an abstract module, or interface, is implemented by a concrete instantiation. In the context of [6], our programming language consists of a standard one, plus abstract module operations that are guaranteed to satisfy some specification. We wish to show that, given concrete and abstract modules, and a relation relating their equivalent states, any execution of the program that can happen when using the concrete module can also happen when using the abstract one.

We simplify the data refinement theory by eliminating the need for two somewhat unintuitive requirements used in [6], called contents independence and growing relations. Contents independence is a strengthening of locality that is implied by the stronger behavior preservation. A growing relation is a technical requirement guaranteeing that the area of memory used by the abstract module is a subset of that used by the concrete one. It turns out that behavior preservation is strong enough to completely eliminate the need to require growing relations, *without* automatically implying that any relations are growing. Therefore, we can prove refinement between some modules (e.g., ones that use completely disjoint areas of memory) that the system of [6] cannot handle.

Our third metatheoretical simplification is in the context of Relational Separation Logic, defined in [14]. Relational Separation Logic is a tool for reasoning about the relationship between two executions on different programs. In [14], soundness of the relational frame rule is initially shown to be dependent on programs being deterministic. The author presents a reasonable solution for making the frame rule sound in the presence of nondeterminism, but the solution is somewhat unintuitive and, more importantly, a significant chunk of the paper (about 9 pages out of 41) is devoted to developing the technical details of the solution. We show that under the context of behavior preservation, the relational frame rule as initially defined is already sound in the presence of nondeterminism, so that section of the paper is no longer needed.

The fourth simplification is minor, but still worth noting. For technical reasons, the standard definition of locality does not play well with a model in which the total amount of available memory is finite. Separation Logic generally avoids this issue by simply using an infinite space of memory. This works fine, but there may be situations in which we wish to use a model that more closely represents what is actually going on inside our computer. While Separation Logic can be made to work in the presence of finite memory, doing so is not a trivial matter. We will show that under our stronger notion of locality, no special treatment is required for finite-sized models.

All proofs in Sections 3 and 4 have been fully mechanized in the Coq proof assistant [7]. The Coq source files, along with their conversions to pdf, can be found at the link to the technical report for this paper [5].

2 Locality and Behavior Preservation

In standard Separation Logic [8, 13, 15, 4], there are two locality properties, known as Safety Monotonicity and the Frame Property, that together imply soundness of the frame rule. Safety Monotonicity says that any time a program executes safely in a certain state, the same program must also execute safely in any larger state — in other words, unused resources cannot cause a program to crash. The Frame Property says that if a program executes safely on a small state, then any terminating execution of the program on a larger state can be tracked back to some terminating execution on the small state by assuming that the extra added state has no effect and is unchanged. Furthermore, there is a

third property, called Termination Monotonicity, that is required whenever we are interested in reasoning about divergence (nontermination). This property says that if a program executes safely and never diverges on a small state, then it cannot diverge on any larger state.

To describe these properties formally, we first formalize the idea of program state. We will describe the theory somewhat informally here; full formal detail will be described later in Section 4. We define states σ to be members of an abstract set Σ. We assume that whenever two states σ_0 and σ_1 are "disjoint," written $\sigma_0 \# \sigma_1$, they can be combined to form the larger state $\sigma_0 \cdot \sigma_1$. Intuitively, two states are disjoint when they occupy disjoint areas of memory.

We represent the semantic meaning of a program C by a binary relation $[\![C]\!]$. We use the common notational convention aRb for a binary relation R to denote $(a, b) \in R$. Intuitively, $\sigma [\![C]\!] \sigma'$ means that, when executing C on initial state σ, it is possible to terminate in state σ'. Note that if σ is related by $[\![C]\!]$ to more than one state, this simply means that C is a nondeterministic program.

We also define two special behaviors `bad` and `div`:

- The notation $\sigma [\![C]\!] \texttt{bad}$ means that C can crash or get stuck when executed on σ, while
- The notation $\sigma [\![C]\!] \texttt{div}$ means that C can diverge (execute forever) when executed on σ.

As a notational convention, we use τ to range over elements of $\Sigma \cup \{\texttt{bad}, \texttt{div}\}$. We require that for any state σ and program C, there is always at least one τ such that $\sigma [\![C]\!] \tau$. In other words, every execution must either crash, go on forever, or terminate in some state.

Now we can define the properties described above more formally. Following are definitions of Safety Monotonicity, the Frame Property, and Termination Monotonicity, respectively:

1.) $\neg \sigma_0 [\![C]\!] \texttt{bad} \wedge \sigma_0 \# \sigma_1 \implies \neg (\sigma_0 \cdot \sigma_1) [\![C]\!] \texttt{bad}$

2.) $\neg \sigma_0 [\![C]\!] \texttt{bad} \wedge (\sigma_0 \cdot \sigma_1) [\![C]\!] \sigma' \implies \exists \sigma_0' \,.\, \sigma' = \sigma_0' \cdot \sigma_1 \wedge \sigma_0 [\![C]\!] \sigma_0'$

3.) $\neg \sigma_0 [\![C]\!] \texttt{bad} \wedge \neg \sigma_0 [\![C]\!] \texttt{div} \wedge \sigma_0 \# \sigma_1 \implies \neg (\sigma_0 \cdot \sigma_1) [\![C]\!] \texttt{div}$

The standard definition of locality was defined in this way because it is the minimum requirement needed to make the frame rule sound — it is as weak as it can possibly be without breaking the logic. It was not defined to correspond with any intuitive notion of locality. As a result, there are two subtleties in the definition that might seem a bit odd. We will now describe these subtleties and the changes we make to get rid of them. Note that we are not arguing in this section that there is any benefit to changing locality in this way (other than the arguably vacuous benefit of corresponding to our "intuition" of locality) — the benefit will become clear when we discuss how our change simplifies the metatheory in Section 5.

The first subtlety is that Termination Monotonicity only applies in one direction. This means that we could have a program C that runs forever on a

state σ, but when we add unused state, we suddenly lose the ability for that infinite execution to occur. We can easily get rid of this subtlety by replacing Termination Monoticity with the following Termination Equivalence property:

$$\neg\sigma_0[\![C]\!]\texttt{bad} \wedge \sigma_0\#\sigma_1 \implies (\sigma_0[\![C]\!]\texttt{div} \iff (\sigma_0 \cdot \sigma_1)[\![C]\!]\texttt{div})$$

The second subtlety is that locality gives us a way of tracking an execution on a large state back to a small one, but it does not allow for the other way around. This means that there can be an execution on a state σ that becomes invalid when we add unused state. This subtlety is a little trickier to remedy than the other. If we think of the Frame Property as really being a "Backwards Frame Property," in the sense that it only works in the direction from large state to small state, then we clearly need to require a corresponding Forwards Frame Property. We would like to say that if C takes σ_0 to σ_0' and we add the unused state σ_1, then C takes $\sigma_0 \cdot \sigma_1$ to $\sigma_0' \cdot \sigma_1$:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0\#\sigma_1 \implies (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

Unfortunately, there is no guarantee that $\sigma_0' \cdot \sigma_1$ is defined, as the states might not occupy disjoint areas of memory. In fact, if C causes our initial state to grow, say by allocating memory, then there will always be some σ_1 that is disjoint from σ_0 but not from σ_0' (e.g., take σ_1 to be exactly that allocated memory). Therefore, it seems as if we are doomed to lose behavior in such a situation upon adding unused state.

There is, however, a solution worth considering: we could disallow programs from ever increasing state. In other words, we can require that whenever C takes σ_0 to σ_0', the area of memory occupied by σ_0' must be a subset of that occupied by σ_0. In this way, anything that is disjoint from σ_0 must also be disjoint from σ_0', so we will not lose any behavior. Formally, we express this property as:

$$\sigma_0[\![C]\!]\sigma_0' \implies (\forall\sigma_1 \,.\, \sigma_0\#\sigma_1 \Rightarrow \sigma_0'\#\sigma_1)$$

We can conveniently combine this property with the previous one to express the Forwards Frame Property as the following condition:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0\#\sigma_1 \implies \sigma_0'\#\sigma_1 \wedge (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

At first glance, it may seem imprudent to impose this requirement, as it apparently disallows memory allocation. However, it is in fact still possible to model memory allocation — we just have to be a little clever about it. Specifically, we can include a set of memory locations in our state that we designate to be the "free list[1]." When memory is allocated, all allocated cells must be taken from the free list. Contrast this to standard Separation Logic, in which newly-allocated heap cells are taken from outside the state. In the next section, we will show that we can add a free list in this way to the model of Separation Logic without requiring a change to any of the inference rules.

We conclude this section with a brief justification of the term "behavior preservation." Given that C runs safely on a state σ_0, we think of a behavior of C on

[1] The free list is actually a set rather than a list; we use the term "free list" because it is commonly used in the context of memory allocation.

$$
\begin{array}{rl}
E & ::= E + E' \mid E - E' \mid E \times E' \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x \mid y \mid \ldots \\
B & ::= E = E' \mid \texttt{false} \mid B \Rightarrow B' \\
P, Q & ::= B \mid \texttt{false} \mid \texttt{emp} \mid E \mapsto E' \mid P \Rightarrow Q \mid \forall x.P \mid P * Q \\
C & ::= \texttt{skip} \mid x := E \mid x := [E] \mid [E] := E' \\
 & \quad \mid x := \texttt{cons}(E_1, \ldots, E_n) \mid \texttt{free}(E) \mid C; C' \\
 & \quad \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C' \mid \texttt{while}\, B\, \texttt{do}\, C
\end{array}
$$

Fig. 1. Assertion and Program Syntax

σ_0 as a particular execution, which can either diverge or terminate at some state σ_0'. The Forwards Frame Property tells us that execution on a larger state $\sigma_0 \cdot \sigma_1$ simulates execution on the smaller state σ_0, while the Backwards (standard) Frame Property says that execution on the smaller state simulates execution on the larger one. Since standard locality only requires simulation in one direction, it is possible for a program to have fewer valid executions, or behaviors, when executing on $\sigma_0 \cdot \sigma_1$ as opposed to just σ_0. Our stronger locality disallows this from happening, enforcing a bisimulation under which all behaviors are preserved when extra resources are added.

3 Impact on a Concrete Separation Logic

We will now present one possible RAM model that enforces our stronger notion of locality without affecting the inference rules of standard Separation Logic. In the standard model of [13], a program state consists of two components: a variable store and a heap. When new memory is allocated, the memory is "magically" added to the heap. As shown in Section 2, we cannot allow allocation to increase the program state in this way. Instead, we will include an explicit free list, or a set of memory locations available for allocation, inside of the program state. Thus a state is now is a triple (s, h, f) consisting of a store, a heap, and a free list, with the heap and free list occupying disjoint areas of memory. Newly-allocated memory will always come from the free list, while deallocated memory goes back into the free list. Since the standard formulation of Separation Logic assumes that memory is infinite and hence that allocation never fails, we similarly require that the free list be infinite. More specifically, we require that there is some location n such that all locations above n are in the free list.

Formally, states are defined as follows:

$$\text{Var } V \triangleq \{x, y, z, \ldots\} \quad \text{Store } S \triangleq V \to \mathbb{Z} \quad \text{Heap } H \triangleq \mathbb{N} \underset{\text{fin}}{\rightharpoonup} \mathbb{Z}$$

$$\text{Free List } F \triangleq \{N \in \mathcal{P}(\mathbb{N}) \mid \exists n \,.\, \forall k \geq n \,.\, k \in N\}$$

$$\text{State } \Sigma \triangleq \{(s, h, f) \in S \times H \times F \mid \text{dom}(h) \cap f = \emptyset\}$$

As a point of clarification, we are not claiming here that including the free list in the state model is a novel idea. Other systems (e.g., [12]) have made use of

a very similar idea. The two novel contributions that we will show in this section are: (1) that a state model which includes an explicit free list can provide a behavior-preserving semantics, and (2) that the corresponding program logic can be made to be completely backwards-compatible with standard Separation Logic (meaning that any valid Separation Logic derivation is also a valid derivation in our logic).

Assertion syntax and program syntax are given in Figure 1, and are exactly the same as in the standard model for Separation Logic.

Our satisfaction judgement $(s, h, f) \models P$ for an assertion P is defined by ignoring the free list and only considering whether (s, h) satisfies P. Our definition of $(s, h) \models P$ is identical to that of standard Separation Logic.

The small-step operational semantics for our machine is defined as $\sigma, C \longrightarrow \sigma', C'$ and is straightforward; the full details can be found in the extended TR. The most interesting aspects are the rules for allocation and deallocation, since they make use of the free list. $x := \texttt{cons}(E_1, \ldots, E_n)$ allocates a nondeterministically-chosen contiguous block of n heap cells from the free list, while $\texttt{free}(E)$ puts the single heap cell pointed to by E back onto the free list. None of the operations make use of any memory existing outside the program state — this is the key for obtaining behavior-preservation.

To see how out state model fits into the structure defined in Section 2, we need to define the state combination operator. Given two states $\sigma_1 = (s_1, h_1, f_1)$ and $\sigma_2 = (s_2, h_2, f_2)$, the combined state $\sigma_1 \cdot \sigma_2$ is equal to $(s_1, h_1 \uplus h_2, f_1)$ if $s_1 = s_2$, $f_1 = f_2$, and the domains of h_1 and h_2 are disjoint; otherwise, the combination is undefined. Note that this combined state satisfies the requisite condition $\mathrm{dom}(h_1 \uplus h_2) \cap f_1 = \emptyset$ because h_1, h_2, and f_1 are pairwise disjoint by assumption. The most important aspect of this definition of state combination is that we can never change the free list when adding extra resources. This guarantees behavior preservation of the nondeterministic memory allocator because the allocator's set of possible behaviors is precisely defined by the free list.

In order to formally compare our logic to "standard" Separation Logic, we need to provide the standard version of the small-step operational semantics, denoted as $(s, h), C \rightsquigarrow (s', h'), C'$. This semantics does not have explicit free lists in the states, but instead treats all locations outside the domain of h as free. We formalize this semantics in the extended TR, and prove the following relationship between the two operational semantics:

$$(s, h), C \overset{n}{\rightsquigarrow} (s', h'), C' \iff \exists f, f' \,.\, (s, h, f), C \xrightarrow{n} (s', h', f'), C'$$

The inference rules in the form $\vdash \{P\}\, C\, \{Q\}$ for our logic are same as those used in standard Separation Logic. In the extended TR, we state all the inference rules and prove that our logic is both sound and complete; therefore, behavior preservation does not cause any complications in the usage of Separation Logic. Any specification that can be proved using the standard model can also be proved using our model. Also in the TR, we prove that our model enjoys the stronger, behavior-preserving notion of locality described in Sec 2.

Even though our logic works exactly the same as standard Separation Logic, our underlying model now has this free list within the state. Therefore, if we

so desire, we could define additional assertions and inference rules allowing for more precise reasoning involving the free list. One idea is to have a separate, free list section of assertions in which we write, for example, $E * \texttt{true}$ to claim that E is a part of the free list. Then the axiom for `free` would look like:

$$\{E \mapsto -; \texttt{true}\}\, \texttt{free}(E)\, \{\texttt{emp}; E * \texttt{true}\}$$

4 The Abstract Logic

In order to clearly explain how our stronger notion of locality resolves the metatheoretical issues described in Section 1, we will first formally describe how our locality fits into a context similar to that of Abstract Separation Logic [4]. With a minor amount of work, the logic of Section 3 can be molded into a particular instance of the abstract logic presented here.

We define a *separation algebra* to be a set of states Σ, along with a partial associative and commutative operator $\cdot : \Sigma \to \Sigma \rightharpoonup \Sigma$. The disjointness relation $\sigma_0 \# \sigma_1$ holds iff $\sigma_0 \cdot \sigma_1$ is defined, and the substate relation $\sigma_0 \preceq \sigma_1$ holds iff there is some σ_0' such that $\sigma_0 \cdot \sigma_0' = \sigma_1$. A particular element of Σ is designated as a unit state, denoted u, with the property that for any σ, $\sigma \# u$ and $\sigma \cdot u = \sigma$. We require the $\cdot$ operator to be cancellative, meaning that $\sigma \cdot \sigma_0 = \sigma \cdot \sigma_1 \Rightarrow \sigma_0 = \sigma_1$.

An *action* is a set of pairs of type $\Sigma \cup \{\texttt{bad}, \texttt{div}\} \times \Sigma \cup \{\texttt{bad}, \texttt{div}\}$. We require the following two properties: (1) actions always relate `bad` to `bad` and `div` to `div`, and never relate `bad` or `div` to anything else; and (2) actions are total, in the sense that for any τ, there exists some τ' such that $\tau A \tau'$ (recall from Section 2 that we use τ to range over elements of $\Sigma \cup \{\texttt{bad}, \texttt{div}\}$). Note that these two requirements are preserved over the standard composition of relations, as well as over both finitary and infinite unions. We write `Id` to represent the identity action $\{(\tau, \tau) \mid \tau \in \Sigma \cup \{\texttt{bad}, \texttt{div}\}\}$.

Note that it is more standard in the literature to have the domain of actions range only over Σ — we use $\Sigma \cup \{\texttt{bad}, \texttt{div}\}$ here because it has the pleasant effect of making $[\![C_1; C_2]\!]$ correspond precisely to standard composition. Intuitively, once an execution goes wrong, it continues to go wrong, and once an execution diverges, it continues to diverge.

A *local action* is an action A that satisfies the following four properties, which respectively correspond to Safety Monotonicity, Termination Equivalence, the Forwards Frame Property, and the Backwards Frame Property from Section 2:

1.) $\neg \sigma_0 A \texttt{bad} \wedge \sigma_0 \# \sigma_1 \implies \neg(\sigma_0 \cdot \sigma_1) A \texttt{bad}$
2.) $\neg \sigma_0 A \texttt{bad} \wedge \sigma_0 \# \sigma_1 \implies (\sigma_0 A \texttt{div} \iff (\sigma_0 \cdot \sigma_1) A \texttt{div})$
3.) $\sigma_0 A \sigma_0' \wedge \sigma_0 \# \sigma_1 \implies \sigma_0' \# \sigma_1 \wedge (\sigma_0 \cdot \sigma_1) A (\sigma_0' \cdot \sigma_1)$
4.) $\neg \sigma_0 A \texttt{bad} \wedge (\sigma_0 \cdot \sigma_1) A \sigma' \implies \exists \sigma_0' \,.\, \sigma' = \sigma_0' \cdot \sigma_1 \wedge \sigma_0 A \sigma_0'$

We denote the set of all local actions by **LocAct**. We now show that the set of local actions is closed under composition and (possibly infinite) union. We use

$$C \ ::= c \mid C_1; C_2 \mid C_1 + C_2 \mid C^*$$

$$\forall c \,.\, [\![c]\!] \in \mathbf{LocAct} \qquad [\![C_1; C_2]\!] \triangleq [\![C_1]\!]; [\![C_2]\!]$$

$$[\![C_1 + C_2]\!] \triangleq [\![C_1]\!] \cup [\![C_2]\!] \qquad [\![C^*]\!] \triangleq \bigcup_{n \in \mathbb{N}} [\![C]\!]^n$$

$$[\![C]\!]^0 \triangleq \mathtt{Id} \qquad [\![C]\!]^{n+1} \triangleq [\![C]\!]; [\![C]\!]^n$$

Fig. 2. Command Definition and Denotational Semantics

the notation $A_1; A_2$ to denote composition, and $\bigcup \mathcal{A}$ to denote union (where $\mathcal{A}$ is a possibly infinite set of actions). The formal definitions of these operations follow. Note that we require that $\mathcal{A}$ be non-empty. This is necessary because $\bigcup \emptyset$ is $\emptyset$, which is not a valid action. Unless otherwise stated, whenever we write $\bigcup \mathcal{A}$, there will always be an implicit assumption that $\mathcal{A} \neq \emptyset$.

$$\tau A_1; A_2 \tau' \iff \exists \tau'' \,.\, \tau A_1 \tau'' \wedge \tau'' A_2 \tau'$$
$$\tau \bigcup \mathcal{A} \tau' \iff \exists A \in \mathcal{A} \,.\, \tau A \tau' \qquad (\mathcal{A} \neq \emptyset)$$

Lemma 1. *If A_1 and A_2 are local actions, then $A_1; A_2$ is a local action.*

Lemma 2. *If every A in the set $\mathcal{A}$ is a local action, then $\bigcup \mathcal{A}$ is a local action.*

Figure 2 defines our abstract program syntax and semantics. The language consists of primitive commands, sequencing $(C_1; C_2)$, nondeterministic choice $(C_1 + C_2)$, and finite iteration (C^*). The semantics of primitive commands are abstracted — the only requirement is that they are local actions. Therefore, from the two previous lemmas and the trivial fact that `Id` is a local action, it is clear that the semantics of *every* program is a local action.

Note that in our concrete language used if statements and while loops. As shownin [4], it is possible to represent if and while constructs with finite iteration and nondeterministic choice by including a primitive command $\mathtt{assume}(B)$, which does nothing if the boolean expression B is true, and diverges otherwise.

Now that we have defined the interpretation of programs as local actions, we can talk about the meaning of a triple $\{P\}\, C\, \{Q\}$. We define an assertion P to be a set of states, and we say that a state σ satisfies P iff $\sigma \in P$. We can then define the separating conjunction as follows:

$$P * Q \triangleq \{\sigma \in \Sigma \mid \exists \sigma_0 \in P, \sigma_1 \in Q \,.\, \sigma = \sigma_0 \cdot \sigma_1\}$$

Given an assignment of primitive commands to local actions, we say that a triple is valid, written $\models \{P\}\, C\, \{Q\}$, just when the following two properties hold

$$\frac{\neg\sigma[\![c]\!]\texttt{bad}}{\vdash \{\{\sigma\}\}\, c\, \{\{\sigma' \mid \sigma[\![c]\!]\sigma'\}\}}\ (\text{PRIM}) \qquad \frac{\vdash \{P\}\, C_1\, \{Q\} \qquad \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1; C_2\, \{R\}}\ (\text{SEQ})$$

$$\frac{\vdash \{P\}\, C_1\, \{Q\} \qquad \vdash \{P\}\, C_2\, \{Q\}}{\vdash \{P\}\, C_1 + C_2\, \{Q\}}\ (\text{PLUS}) \qquad \frac{\vdash \{P\}\, C\, \{P\}}{\vdash \{P\}\, C^*\, \{P\}}\ (\text{STAR})$$

$$\frac{\vdash \{P\}\, C\, \{Q\}}{\vdash \{P * R\}\, C\, \{Q * R\}}\ (\text{FRAME}) \qquad \frac{P' \subseteq P \qquad \vdash \{P\}\, C\, \{Q\} \qquad Q \subseteq Q'}{\vdash \{P'\}\, C\, \{Q'\}}\ (\text{CONSEQ})$$

$$\frac{\forall i \in I\ .\ \vdash \{P_i\}\, C\, \{Q_i\}}{\vdash \{\bigcup P_i\}\, C\, \{\bigcup Q_i\}}\ (\text{DISJ}) \qquad \frac{\forall i \in I\ .\ \vdash \{P_i\}\, C\, \{Q_i\} \qquad I \neq \emptyset}{\vdash \{\bigcap P_i\}\, C\, \{\bigcap Q_i\}}\ (\text{CONJ})$$

Fig. 3. Inference Rules

for all states σ and σ':

$$1.)\ \sigma \in P \implies \neg\sigma[\![C]\!]\texttt{bad}$$
$$2.)\ \sigma \in P \wedge \sigma[\![C]\!]\sigma' \implies \sigma' \in Q$$

The inference rules of the logic are given in Figure 3. Note that we are taking a significant presentation shortcut here in the inference rule for primitive commands. Specifically, we assume that we know the exact local action $[\![c]\!]$ of each primitive command c. This assumption makes sense when we define our own primitive commands, as we do in the logic of Section 3. However, in a more general setting, we might be provided with an opaque function along with a specification (precondition and postcondition) for the function. Since the function is opaque, we must consider it to be a primitive command in the abstract setting. Yet we do not know how it is implemented, so we do not know its precise local action. In [4], the authors provide a method for inferring a "best" local action from the function's specification. With a decent amount of technical development, we can do something similar here, using our stronger definition of locality. These details can be found in the technical report [5].

Given this assumption, we prove soundness and completeness of our abstract logic. The details of the proof can be found in our Coq implementation [5].

Theorem 1 (Soundness and Completeness).

$$\vdash \{P\}\, C\, \{Q\} \iff \models \{P\}\, C\, \{Q\}$$

5 Simplifying Separation Logic Metatheory

Now that we have an abstracted formalism of our behavior-preserving local actions, we will resolve each of the four metatheoretical issues described in Sec 1.

5.1 Footprints and Smallest Safe States

Consider a situation in which we are handed a program C along with a specification of what this program does. The specification consists of a set of axioms; each axiom has the form $\{P\}\, C\, \{Q\}$ for some precondition P and postcondition Q. A common question to ask would be: is this specification *complete*? In other words, if the triple $\models \{P\}\, C\, \{Q\}$ is valid for some P and Q, then is it possible to derive $\vdash \{P\}\, C\, \{Q\}$ from the provided specification?

In standard Separation Logic, it can be extremely difficult to answer this question. In [12], the authors conduct an in-depth study of various conditions and circumstances under which it is possible to prove that certain specifications are complete. However, in the general case, there is no easy way to prove this.

We can show that under our assumption of behavior preservation, there is a very easy way to guarantee that a specification is complete. In particular, a specification that describes the exact behavior of C on all of its *smallest safe states* is always complete. Formally, a smallest safe state is a state σ such that $\neg\sigma[\![C]\!]\texttt{bad}$ and, for all $\sigma' \prec \sigma$, $\sigma'[\![C]\!]\texttt{bad}$.

To see that such a specification may not be complete in standard Separation Logic, we borrow an example from [12]. Consider the program C, defined as $x := \texttt{cons}(0); \texttt{free}(x)$. This program simply allocates a single cell and then frees it. Under the standard model, the smallest safe states are those of the form $(s, \emptyset)$ for any store s. For simplicity, assume that the only variables in the store are x and y. Define the specification to be the infinite set of triples that have the following form, for any a, b in $\mathbb{Z}$, and any a' in $\mathbb{N}$:

$$\{x = a \wedge y = b \wedge \texttt{emp}\}\, C\, \{x = a' \wedge y = b \wedge \texttt{emp}\}$$

Note that a' must be in $\mathbb{N}$ because only valid unallocated memory addresses can be assigned into x. It should be clear that this specification describes the exact behavior on all smallest safe states of C. Now we claim that the following triple is valid, but there is no way to derive it from the specification.

$$\{x = a \wedge y = b \wedge y \mapsto -\}\, C\, \{x = a' \wedge y = b \wedge y \mapsto - \wedge a' \neq b\}$$

The triple is clearly valid because a' must be a memory address that was initially unallocated, while address b was initially allocated. Nevertheless, there will not be any way to derive this triple, even if we come up with new assertion syntax or inference rules. The behavior of C on the larger state is different from the behavior on the small one, but there is no way to recover this fact once we make C opaque. It can be shown (see [12]) that if we add triples of the above form to our specification, then we will obtain a complete specification for C. Yet there is no straightforward way to see that such a specification is complete.

We will now formally prove that, in our system, there is a canonical form for complete specification. We first note that we will need to assume that our set of states is well-founded with respect to the substate relation (i.e., there is no infinite strictly-decreasing chain of states). This assumption is true for

most standard models of Separation Logic, and furthermore, there is no reason to intuitively believe that the smallest safe states should be able to provide a complete specification when the assumption is not true.

We say that a specification Ψ is *complete for* C if, whenever $\models \{P\}\, C\, \{Q\}$ is valid, the triple $\vdash \{P\}\, C\, \{Q\}$ is derivable using only the inference rules that are not specific to the structure of C (i.e., the frame, consequence, disjunction, and conjunction rules), plus the following axiom rule:

$$\frac{\{P\}\, C\, \{Q\} \in \Psi}{\vdash \{P\}\, C\, \{Q\}}$$

For any σ, let $\sigma[\![C]\!]$ denote the set of all σ' such that $\sigma[\![C]\!]\sigma'$. For any set of states S, we define a *canonical specification on* S as the set of triples of the form $\{\{\sigma\}\}\, C\, \{\sigma[\![C]\!]\}$ for any state $\sigma \in S$. If there exists a canonical specification on S that is complete for C, then we say that S forms a *footprint* for C. We can then prove the following theorem (see the extended TR):

Theorem 2. *For any program C, the set of all smallest safe states of C forms a footprint for C.*

Note that while this theorem guarantees that the canonical specification is complete, we may not actually be able to write down the specification simply because the assertion language is not expressive enough. This would be the case for the behavior-preserving nondeterministic memory allocator if we used the assertion language presented in Section 3. We could, however, express canonical specifications in that system by extending the assertion language to talk about the free list (as briefly discussed at the end of Section 3).

5.2 Data Refinement

In [6], the goal is to formalize the concept of having a concrete module correctly implement an abstract one, within the context of Separation Logic. Specifically, the authors prove that as long as a client program "behaves nicely," any execution of the program using the concrete module can be tracked to a corresponding execution using the abstract module. The client states in the corresponding executions are identical, so the proof shows that a well-behaved client cannot tell the difference between the concrete and abstract modules.

To get their proof to work out, the authors require two somewhat odd properties to hold. The first is called *contents independence*, and is an extra condition on top of the standard locality conditions. The second is called a *growing relation* — it refers to the relation connecting a state of the abstract module to its logically equivalent state(s) in the concrete module. All relations connecting the abstract and concrete modules in this way are required to be growing, which means that the domain of memory used by the abstract state must be a subset of that used by the concrete state. This is a somewhat unintuitive and restrictive requirement which is needed for purely technical reasons. We will show that

behavior preservation completely eliminates the need for both contents independence and growing relations.

We now provide a formal setting for the data refinement theory. This formal setting is similar to the one in [6], but we will make some minor alterations to simplify the presentation. The programming language is defined as:

$$\begin{aligned} C \ ::= & \ \texttt{skip} \mid c \mid \texttt{m} \mid C_1; C_2 \mid \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2 \\ & \mid \texttt{while}\, B\, \texttt{do}\, C \end{aligned}$$

c is a primitive command (sometimes referred to as "client operation" in this context). $\texttt{m}$ is a *module command* taken from an abstracted set **MOp** (e.g., a memory manager might implement the two module commands `cons` and `free`).

The abstracted client and module commands are assumed to have a semantics mapping them to particular local actions. We of course use our behavior-preserving notion of "local" here, whereas in [6], the authors use the three properties of safety monotonicity, the (backwards) frame property, and a new property called contents independence. It is trivial to show that behavior preservation implies contents independence, as contents independence is essentially a forwards frame property that can only be applied under special circumstances.

A *module* is a pair (p, η) representing a particular implementation of the module commands in **MOp**; the state predicate p describes the module's *invariant* (e.g., that a valid free list is stored starting at a location pointed to by a particular head pointer), while η is a function mapping each module command to a particular local action. The predicate p is required to be *precise* [11], meaning that no state can have more than one substate satisfying p (if a state σ does have a substate satisfying p, then we refer to that uniquely-defined state as σ_p). Additionally, all module operations are required to preserve the invariant p:

$$\neg\sigma(\eta\texttt{m})\texttt{bad} \wedge \sigma \in p * \texttt{true} \wedge \sigma(\eta\texttt{m})\sigma' \Longrightarrow \sigma' \in p * \texttt{true}$$

We define a big-step operational semantics parameterized by a module (p, η). This semantics is fundamentally the same as the one defined in [6]; the extended TR contains the full details. The only aspect that is important to mention here is that the semantics is equipped with a special kind of faulting called "access violation." Intuitively, an access violation occurs when a client operation's execution depends on the module's portion of memory. More formally, it occurs when the client operation executes safely on a state where the module's memory is present (i.e., a state satisfying $p * \texttt{true}$), but faults when that memory is removed from the state.

The main theorem that we get out of this setup is a refinement simulation between a program being run in the presence of an abstract module (p, η), and the same program being run in the presence of a concrete module (q, μ) that implements the same module commands (i.e., $\lfloor \eta \rfloor = \lfloor \mu \rfloor$, where the floor notation indicates domain). Suppose we have a binary relation R relating states of the abstract module to those of the concrete module. For example, if our modules are memory managers, then R might relate a particular set of memory locations

available for allocation to all lists containing that set of locations in some order. To represent that R relates abstract module states to concrete module states, we require that whenever $\sigma_1 R \sigma_2$, $\sigma_1 \in p$ and $\sigma_2 \in q$. Given this relation R, we can make use of the separating conjunction of Relational Separation Logic [14] and write $R * \texttt{Id}$ to indicate the relation relating any two states of the form $\sigma_p \cdot \sigma_c$ and $\sigma_q \cdot \sigma_c$, where $\sigma_p R \sigma_q$.

Now, for any module (p, η), let $C[(p, \eta)]$ be notation for the program C whose semantics have (p, η) filled in for the parameter module. Then our main theorem says that, if $\eta(f)$ simulates $\mu(f)$ under relation $R * \texttt{Id}$ for all $f \in \lfloor \eta \rfloor$, then for any program C, $C[(p, \eta)]$ also simulates $C[(q, \mu)]$ under relation $R * \texttt{Id}$. More formally, say that C_1 simulates C_2 under relation R (written $R; C_2 \subseteq C_1; R$) when, for all σ_1, σ_2 such that $\sigma_1 R \sigma_2$:

1.) $\sigma_1 [\![C_1]\!] \texttt{bad} \iff \sigma_2 [\![C_2]\!] \texttt{bad}$, and
2.) $\neg\, \sigma_1 [\![C_1]\!] \texttt{bad} \implies (\forall \sigma_2' \,.\, \sigma_2 [\![C_2]\!] \sigma_2' \Rightarrow \exists \sigma_1' \,.\, \sigma_1 [\![C_1]\!] \sigma_1' \wedge \sigma_1' R \sigma_2')$

Theorem 3. *Suppose we have modules (p, η) and (q, μ) with $\lfloor \eta \rfloor = \lfloor \mu \rfloor$ and a refinement relation R as described above, such that $R * \texttt{Id}; \mu(f) \subseteq \eta(f); R * \texttt{Id}$ for all $f \in \lfloor \eta \rfloor$. Then, for any program C, $R * \texttt{Id}; C[(q, \mu)] \subseteq C[(p, \eta)]; R * \texttt{Id}$.*

While the full proof can be found in the extended TR, we will semi-formally describe here the one case that highlights why behavior preservation eliminates the need for contents independence and growing relations: when C is simply a client command c. We wish to prove that $C[(p, \eta)]$ simulates $C[(q, \mu)]$, so suppose we have related states σ_1 and σ_2, and executing c on σ_2 results in σ_2'. Since σ_1 and σ_2 are related by $R * \texttt{Id}$, we have that $\sigma_1 = \sigma_p \cdot \sigma_c$ and $\sigma_2 = \sigma_q \cdot \sigma_c$. We know that (1) $\sigma_q \cdot \sigma_c \xrightarrow{c} \sigma_2'$, (2) c is local, and (3) c runs safely on σ_c because the client operation's execution must be independent of the module state σ_q; thus the backwards frame property tells us that $\sigma_2' = \sigma_q \cdot \sigma_c'$ and $\sigma_c \xrightarrow{c} \sigma_c'$. Now, if c is behavior-preserving, then we can simply apply the forwards frame property, framing on the state σ_p, to get that $\sigma_p \# \sigma_c'$ and $\sigma_p \cdot \sigma_c \xrightarrow{c} \sigma_p \cdot \sigma_c'$, completing the proof for this case. However, without behavior preservation, contents independence and growing relations are used in [6] to finish the proof. Specifically, because we know that $\sigma_q \cdot \sigma_c \xrightarrow{c} \sigma_q \cdot \sigma_c'$ and that c runs safely on σ_c, contents independence says that $\sigma \cdot \sigma_c \xrightarrow{c} \sigma \cdot \sigma_c'$ for any σ whose domain is a subset of the domain of σ_q. Therefore, we can choose $\sigma = \sigma_p$ because R is a growing relation.

5.3 Relational Separation Logic

Relational Separation Logic [14] allows for simple reasoning about the relationship between two executions. Instead of deriving triples $\{P\}\, C\, \{Q\}$, a user of the logic derives *quadruples* of the form:

$$\{R\} \begin{matrix} C \\ C' \end{matrix} \{S\}$$

R and S are binary relations on states, rather than unary predicates. Semantically, a quadruple says that if we execute the two programs in states that are related by R, then both executions are safe, and any termination states will be related by S. Furthermore, we want to be able to use this logic to prove program equivalence, so we also require that initial states related by R have the same divergence behavior. Formally, we say that the above quadruple is valid if, for any states σ_1, σ_2, σ_1', σ_2':

$$\begin{aligned}
&1.)\ \sigma_1 R \sigma_2 \implies \neg\sigma_1[\![C]\!]\texttt{bad} \wedge \neg\sigma_2[\![C']\!]\texttt{bad} \\
&2.)\ \sigma_1 R \sigma_2 \implies (\sigma_1[\![C]\!]\texttt{div} \iff \sigma_2[\![C']\!]\texttt{div}) \\
&3.)\ \sigma_1 R \sigma_2 \wedge \sigma_1[\![C]\!]\sigma_1' \wedge \sigma_2[\![C']\!]\sigma_2' \implies \sigma_1' S \sigma_2'
\end{aligned}$$

Relational Separation Logic extends the separating conjunction to work for relations, breaking related states into disjoint, correspondingly-related pieces:

$$\begin{aligned}
\sigma_1 (R * S) \sigma_2 \iff \exists\ \sigma_{1r},\ \sigma_{1s},\ \sigma_{2r},\ \sigma_{2s}\ . \\
\sigma_1 = \sigma_{1r} \cdot \sigma_{1s} \wedge \sigma_2 = \sigma_{2r} \cdot \sigma_{2s} \wedge \sigma_{1r} R \sigma_{2r} \wedge \sigma_{1s} S \sigma_{2s}
\end{aligned}$$

Just as Separation Logic has a frame rule for enabling local reasoning, Relational Separation Logic has a frame rule with the same purpose. This frame rule says that, given that we can derive the quadruple above involving R, S, C, and C', we can also derive the following quadruple for any relation T:

$$\{R * T\} \begin{matrix} C \\ C' \end{matrix} \{S * T\}$$

In [14], it is shown that the frame rule is sound when all programs are deterministic but it is unsound if nondeterministic programs are allowed, so this frame rule cannot be used when we have a nondeterministic memory allocator.

To deal with nondeterministic programs, a solution is proposed in [14], in which the interpretation of quadruples is strengthened. The new interpretation for a quadruple containing R, S, C, and C' is that, for any σ_1, σ_2, σ_1', σ_2', σ, σ':

$$\begin{aligned}
&1.)\ \sigma_1 R \sigma_2 \implies \neg\sigma_1[\![C]\!]\texttt{bad} \wedge \neg\sigma_2[\![C']\!]\texttt{bad} \\
&2.)\ \sigma_1 R \sigma_2 \wedge \sigma_1 \# \sigma \wedge \sigma_2 \# \sigma' \implies ((\sigma_1 \cdot \sigma)[\![C]\!]\texttt{div} \iff (\sigma_2 \cdot \sigma')[\![C']\!]\texttt{div}) \\
&3.)\ \sigma_1 R \sigma_2 \wedge \sigma_1[\![C]\!]\sigma_1' \wedge \sigma_2[\![C']\!]\sigma_2' \implies \sigma_1' S \sigma_2'
\end{aligned}$$

Note that this interpretation is the same as before, except that the second property is strengthened to say that divergence behavior must be equivalent not only on the initial states, but also on any larger states. It can be shown that the frame rule becomes sound under this stronger interpretation of quadruples.

In our behavior-preserving setting, it is possible to use the simpler interpretation of quadruples without breaking soundness of the frame rule. We could prove this by directly proving frame rule soundness, but instead we will take a shorter route in which we show that, when actions are behavior-preserving, a

quadruple is valid under the first interpretation above if and only if it is valid under the second interpretation — i.e., the two interpretations are the same in our setting. Since the frame rule is sound under the second interpretation, this implies that it will also be sound under the first interpretation.

Clearly, validity under the second interpretation implies validity under the first, since it is a direct strengthening. To prove the inverse, suppose we have a quadruple (involving R, S, C, and C') that is valid under the first interpretation. Properties 1 and 3 of the second interpretation are identical to those of the first, so all we need to show is that Property 2 holds. Suppose that $\sigma_1 R \sigma_2$, $\sigma_1 \# \sigma$, and $\sigma_2 \# \sigma'$. By Property 1 of the first interpretation, we know that $\neg\sigma_1[\![C]\!]\texttt{bad}$ and $\neg\sigma_2[\![C']\!]\texttt{bad}$. Therefore, Termination Equivalence tells us that $\sigma_1[\![C]\!]\texttt{div} \iff (\sigma_1 \cdot \sigma)[\![C]\!]\texttt{div}$, and that $\sigma_2[\![C']\!]\texttt{div} \iff (\sigma_2 \cdot \sigma')[\![C']\!]\texttt{div}$. Furthermore, we know by Property 2 of the first interpretation that $\sigma_1[\![C]\!]\texttt{div} \iff \sigma_2[\![C']\!]\texttt{div}$. Hence we obtain our desired result.

In case the reader is curious, the reason that the frame rule under the first interpretation is sound when all programs are deterministic is simply that determinism (along with standard locality) implies Termination Equivalence. A proof of this can be found in the extended TR.

5.4 Finite Memory

Since standard locality allows the program state to increase during execution, it does not play nicely with a model in which memory is finite. Consider any command that grows the program state in some way. Such a command is safe on the empty state but, if we extend this empty state to the larger state consisting of all available memory, then the command becomes unsafe. Hence such a command violates Safety Monotonicity.

There is one commonly-used solution for supporting finite memory without enforcing behavior preservation: say that, instead of faulting on the state consisting of all of memory, a state-growing command diverges. Furthermore, to satisfy Termination Monotonicity, we also need to allow the command to diverge on *any* state. The downside of this solution, therefore, is that it is only reasonable when we are not interested in the termination behavior of programs.

When behavior preservation is enforced, we no longer have any issues with finite memory models because program state cannot increase during execution. The initial state is obviously contained within the finite memory, so all states reachable through execution must also be contained within memory.

6 Related Work and Conclusions

The definition of locality (or local action), which enables the frame rule, plays a critical role in Separation Logic [8, 13, 15]. Almost all versions of Separation Logic — including their concurrent [3, 10, 4], higher-order [2], and relational [14] variants, as well as mechanized implementation (e.g., [1]) — have always used

the same locality definition that matches the well-known Safety and Termination Monotonicity properties and the Frame Property [15].

In this paper, we argued a case for strengthening the definition of locality to enforce *behavior preservation*. This means that the behavior of a program when executed on a small state is identical to the behavior when executed on a larger state — put another way, excess, unused state cannot have any effect on program behavior. We showed that this change can be made to have no effect on the usage of Separation Logic, and we gave multiple examples of how it simplifies reasoning about metatheoretical properties.

Determinism Constancy One related work that calls for comparison is the property of "Determinism Constancy" presented by Raza and Gardner [12], which is also a strengthening of locality. While they use a slightly different notion of action than we do, it can be shown that Determinism Constancy, when translated into our context (and ignoring divergence behaviors), is logically equivalent to:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0' \# \sigma_1 \Longrightarrow \sigma_0 \# \sigma_1 \wedge (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

For comparison, we repeat our Forwards Frame Property here:

$$\sigma_0[\![C]\!]\sigma_0' \wedge \sigma_0 \# \sigma_1 \Longrightarrow \sigma_0' \# \sigma_1 \wedge (\sigma_0 \cdot \sigma_1)[\![C]\!](\sigma_0' \cdot \sigma_1)$$

While our strengthening of locality prevents programs from increasing state during execution, Determinism Constancy prevents programs from *decreasing* state. The authors use Determinism Constancy to prove the same property regarding footprints that we proved in Section 5.1. Note that, while behavior preservation does not imply Determinism Constancy, our concrete logic of Section 3 does have the property since it never decreases state (we chose to have the `free` command put the deallocated cell back onto the free list, rather than get rid of it entirely).

While Determinism Constancy is strong enough to prove the footprint property, it does not provide behavior preservation — an execution on a small state can still become invalid on a larger state. Thus it will not, for example, help in resolving the dilemma of growing relations in the data refinement theory. Due to the lack of behavior preservation, we do not expect the property to have a significant impact on the metatheory as a whole. Note, however, that there does not seem to be any harm in using *both* behavior preservation and Determinism Constancy. The two properties together enforce that the area of memory accessible to a program be constant throughout execution.

Module Reasoning Besides our discussion of data refinement in Section 5.2, there has been some previous work on reasoning about modules and their implementations. In [11], a "Hypothetical Frame Rule" is used to allow modular reasoning when a module's implementation is hidden from the rest of the code. In [2], a higher-order frame rule is used to allow reasoning in a higher-order language with hidden module or function code. However, neither of these works discuss relational reasoning between different modules. We are not aware of any relational logic for reasoning about modules.

Acknowledgements. We thank Xinyu Feng and anonymous referees for suggestions and comments on an earlier version of this paper. This material is based on research sponsored by DARPA under agreement numbers FA8750-10-2-0254 and FA8750-12-2-0293, and by NSF grants CNS-0910670, CNS-0915888, and CNS-1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these agencies.

References

1. A. W. Appel and S. Blazy. Separation logic for small-step cminor. In *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 5–21, 2007.
2. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *Proc. 20th IEEE Symp. on Logic in Computer Science*, pages 260–269, 2005.
3. S. Brookes. A semantics for concurrent separation logic. In *Proc. 15th International Conference on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, 2004.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 366–378, July 2007.
5. D. Costanzo and Z. Shao. A case for behavior-preserving actions in separation logic. Technical report, Dept. of Computer Science, Yale University, New Haven, CT, June 2012. `http://flint.cs.yale.edu/publications/bpsl.html`.
6. I. Filipovic, P. W. O'Hearn, N. Torp-Smith, and H. Yang. Blaming the client: on data refinement in the presence of pointers. *Formal Asp. Comput.*, 22(5):547–583, 2010.
7. G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. The Coq release v6.3.1, May 2000.
8. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM Symposium on Principles of Programming Languages*, pages 14–26, Jan. 2001.
9. B. W. Kernighan and D. M. Ritchie. *The C Programming Language (Second Edition)*. Prentice Hall, 1988.
10. P. W. O'Hearn. Resources, concurrency and local reasoning. In *Proc. 15th Int'l Conf. on Concurrency Theory (CONCUR'04)*, volume 3170 of *LNCS*, pages 49–67, 2004.
11. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM Trans. Program. Lang. Syst.*, 31(3):1–50, 2009.
12. M. Raza and P. Gardner. Footprints in local reasoning. *Journal of Logical Methods in Computer Science*, 5(2), 2009.
13. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symp. on Logic in Computer Science*, pages 55–74, July 2002.
14. H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.
15. H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proc. 5th Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.

Modular Verification of Concurrent Thread Management

Yu Guo[1], Xinyu Feng[1], Zhong Shao[2], and Peizhi Shi[1]

[1] University of Science and Technology of China
{guoyu,xyfeng}@ustc.edu.cn sea10197@mail.ustc.edu.cn
[2] Yale University
zhong.shao@yale.edu

Abstract. Thread management is an essential functionality in OS kernels. However, verification of thread management remains a challenge, due to two conflicting requirements: on the one hand, a thread manager—operating below the thread abstraction layer–should hide its implementation details and be verified independently from the threads being managed; on the other hand, the thread management code in many real-world systems is concurrent, which might be executed by the threads being managed, so it seems inappropriate to abstract threads away in the verification of thread managers. Previous approaches on kernel verification view thread managers as sequential code, thus cannot be applied to thread management in realistic kernels. In this paper, we propose a novel two-layer framework to verify concurrent thread management. We choose a lower abstraction level than the previous approaches, where we abstract away the context switch routine only, and allow the rest of the thread management code to run concurrently in the upper level. We also treat thread management data as abstract resources so that threads in the environment can be specified in assertions and be reasoned about in a proof system similar to concurrent separation logic.

1 Introduction

Thread scheduling in modern operating systems provides the functionality of virtualizing processors: when a thread is waiting for an event, it gives the control of the processor to another thread to create the illusion that each thread has its own processor.

Inside a kernel, a thread manager supervises all threads in the system by manipulating data structures called thread control blocks (TCBs). A TCB is used to record important information about a thread, such as the machine context (or processor state), the thread identifier, the status description, the location and size of the stack, the priority for scheduling, and the entry point of thread code. The TCBs are often implemented using data structures such as queues for ready and waiting threads. Clearly, modifying thread queues and TCBs would drastically change the behaviors of threads. Therefore, a correct implementation of thread management is crucial for guaranteeing the whole system safety. Unfortunately, modular verification of real-world thread management code remains a big challenge today.

The challenge comes from two apparently conflicting goals which we want to achieve at the same time: abstraction (for modular verification) and efficiency (for real-world

usability). On the one hand, TCBs, thread queues, and the thread scheduler are specifics used to implement threads so they should sit at a lower abstraction layer. It is natural to abstract them away from threads, and to verify threads and the thread scheduler separately at different abstraction layers. Previous work has shown it is extremely difficult to verify them together in one logic system [15]. On the other hand, in many real-world systems such as Linux-2.6.10 [12] and FreeBSD-5.2 [13], the thread scheduler code itself is also *concurrent* in the sense that there may be multiple threads in the system running the scheduler at the same time. For instance, when a thread invokes a thread scheduler routine (*e.g.*, cleaning up dead threads, load balancing, or thread scheduling) and traverses the thread queue, it may be preempted by other threads who may call the same routine and traverse the queue too. Also, in some systems [12,1] the thread scheduling itself is implemented as a separate thread that runs concurrently with other threads. In these cases, we need to verify thread schedulers in a "multi-threaded" logic, taking threads into account instead of abstracting them away.

Earlier work on thread scheduling verification fails to achieve the two goals at the same time. Ni *et al.* [15] verified both the thread switch and the threads in one logic [14], which treats thread return addresses as first-class code pointers. Although their method may support concurrent thread schedulers in real systems, it loses the abstraction of threads completely, and makes the logic and specifications too complex for practical use. Recent work [3,6] adopts two-layer verification frameworks to verify concurrent kernels. Kernel code is divided into two layers: sequential code in the lower layer and concurrent in the upper layer. In their frameworks, they put the code manipulating TCBs (*e.g.*, thread schedulers) in the low layer, and hide the TCBs of threads in the upper layer so that the threads cannot modify them. Then they use sequential program logics to verify thread management code. However, this approach is not usable for many realistic kernels where thread managers themselves are concurrent and the threads are allowed to modify the TCBs. Other work on OS verification [11,9] only supports non-reentrant kernels, *i.e.*, there is only one thread running in the kernel at any time.

In this paper, we propose a more natural framework to verify concurrent thread managers. Our framework follows the two-layer approach, so concurrent code at the upper layer can be verified modularly with thread abstractions. However, the abstraction level of our framework is much lower than previous frameworks [3,6]. The majority of the code manipulating thread queues and TCBs is put in the upper layer and can be verified as concurrent code. Our framework successfully achieves both verification goals: it not only allows abstraction and modular verification, but also supports concurrency in real-world thread management.

Our work is based on previous work on thread scheduler verification, but makes the following new contributions:

- We introduce a fine-grained abstraction in our two-layer verification framework. The abstraction protects only a small part of sensitive data in TCBs, and at the same time allows multiple threads to modify other part of TCBs safely. Our division of the two abstraction layers is consistent with many real systems. It is more natural and can support more realistic thread managers than previous work.
- In the upper layer, we introduce the idea of treating *threads as resources*. The abstract thread resources can be specified explicitly in the assertion language, and

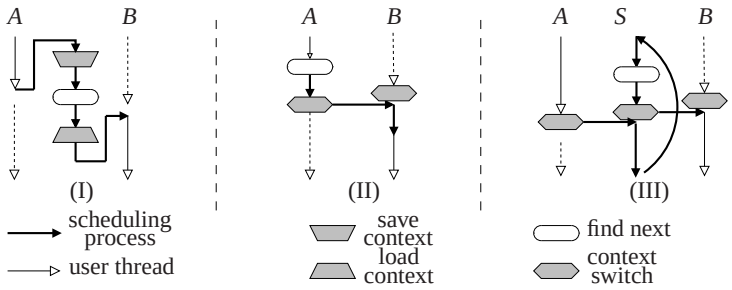


Fig. 1. Three patterns of scheduling

their use by concurrent programs can be reasoned about modularly following concurrent separation logic (CSL) [16]. By enforcing the invariant that the abstract resource is consistent with the concrete thread meta data, we can ensure the safety of the accesses over TCBs and thread queues inside threads.

– Because of the fine-grained abstraction of our approach, the semantics of thread scheduling do not have to be hardwired in the logic. Therefore, our framework can be used to verify various implementation patterns of thread management. We show how to verify the three common patterns of thread scheduling in realistic OS kernels (while previous two-layer frameworks [3,6] can only verify one of them).
– In our extended TR [7], we also use our framework to verify thread schedulers with hardware interrupts, scheduling over multiprocessor with load-balancing, and a set of other thread management routines such as thread creation, join and termination.

The rest of this paper is organized as follows: we first introduce a simplified abstract machine model for the higher-layer of our framework in Sec. 3; to show our main idea, we propose in Sec. 4 our proof system for concurrent thread scheduling code over the abstract machine. We show how to verify two prototypes of schedulers based on context switch in Sec. 5. We compare with related work in Sec. 6, and then conclude.

2 Challenges and our approach

In this section, we illustrate the challenges of verifying code of thread scheduling by showing three patterns of schedulers and discuss the verification issues. Then we informally explain the basic ideas of our approach.

2.1 Three patterns of thread scheduling

By deciding which thread to run next, the thread scheduler is responsible for best utilizing the system and makes multiple threads run concurrently. The scheduling process consists of the following steps: selecting which thread to run next in a thread queue by modifying TCBs, saving the context data of the current thread, and loading the context data of the next thread. Context data is the state of the processor. By saving and loading context data, the processor can run in multiple control flows, *i.e.*, threads. Usually, context data can be saved on stacks or TCBs (we assume in this paper that context

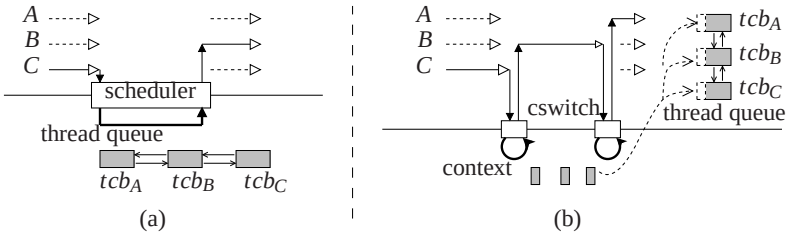


Fig. 2. Abstraction in verification framework

data is saved in TCBs for the brevity of presentation). There are various ways to implement thread schedulers. In Fig. 1 we show three common implementation patterns, all modeled from real systems.

Pattern (I) is popular among embedded OS kernels (*e.g.*, FreeRTOS) and some micro-kernels (*e.g.*, Minix [8] and Exokernel [2]). The scheduler in this pattern is invoked by function calls or interrupts. Thereafter, the scheduling is done in the following steps: (1) saving the current context data, (2) finding the next thread, and (3) loading the context data of the next thread (and switching to it implicitly through function return).

In pattern (II), the scheduling process is a function with the following steps: (1) finding the next thread firstly, (2) performing context switch (saving the current context data, loading the next one, and jumping to the next thread immediately), (3) and running the remaining code of the function when the control is switched back from other threads. This pattern is modeled from some mainstream monolithic kernels (*e.g.*, Linux [12], and FreeBSD). Some embedded kernels (*e.g.*, RTEMS and uClinux) adopt it too. Note that both the involved threads should be allowed to access the thread queue and TCBs when calling the scheduler.

Pattern (III) uses a separate thread, called *scheduler thread*, to do scheduling. One thread may perform scheduling by doing context switch to the scheduler thread. The scheduler thread is a big infinite loop: finding the next thread; performing context switch to the next thread; and looping after return. This pattern can be seen in the GNU-pth thread library, MIT-xv6 kernel, L4::Ka, *etc.*. Similar to pattern (II), all involved threads in this pattern should be allowed to access the TCB of the scheduler thread and the thread queue.

2.2 Challenges

As we can see from the patterns in Fig. 1, the control flow in the scheduling process is very complicated. Threads switch back and forth via manipulating the thread queues and TCBs. It is very natural to share TCBs and the thread queue among threads in order to support all these scheduling patterns. On the other hand, it is important to ensure that the TCBs are accessed in the right way. The system would go wrong if, for instance, a thread erased the context data of another by mistake, or put a dead thread back into the ready thread queue.

To guarantee the safety of the scheduling process, we must fulfill two requirements:

(1) No thread can incorrectly modify the context data in TCBs.

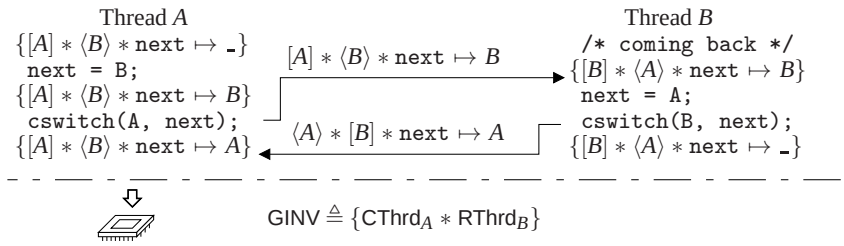


Fig. 3. Abstract thread res. vs. concrete thread res.

(2) The scheduler should know the status of each thread in the thread queues and decide which to run next.

To satisfy the requirement (1), some previous work [3,6] adopts a two-layer-based approach and protects the TCBs through *abstraction*, where the TCBs are simply hidden from kernel threads and become inaccessible. This approach can be used to verify schedulers of pattern (I), for which we show the abstraction line in Fig. 2 (a). Threads above the line cannot modify TCBs, while the scheduler is below this line and has full access to them. The lower-layer scheduler provides an abstract interface to the verification of concurrent thread code at the upper layer. Since it modifies the TCBs in the scheduling time only, we can view the scheduler as a sequential function which does not belong to any thread and can be verified by a conventional Hoare-style logic. However, this approach cannot verify the other two patterns, nor does it fulfill the requirement (2) for concurrent schedulers, where the TCBs are manipulated concurrently (not sequentially as in pattern (I)) and should be known by threads. That is, we cannot completely hide the TCBs from the upper-layer concurrent threads for patterns (II) and (III).

2.3 Our approach

If we inspect the TCB data carefully, we can see that only a small part of the data is crucial to thread behaviors and cannot be accessed concurrently. It is unnecessary to access it concurrently either. The data includes the machine context data and the stack location. We call them *safety-critical* values. Some values can be modified concurrently, but their correctness is still important to the safety of the kernel, *e.g.,* the pointers organizing thread queues and the status field belong to this kind of values. Other values of TCBs have nothing to do with the safety of the kernel and can be modified concurrently definitely, *e.g.,* the name of a thread or debug information.

Lowering the abstraction level. To protect the safety critical part of TCBs, we lower the abstraction line, as shown in Fig. 2 (b). In our framework, the safety-critical data of TCBs is under the abstraction line and hidden from threads. The corresponding operations such as context saving, loading and switching are abstracted away from threads too, with only interfaces exposed to the upper layer. The other part of TCBs are lifted above this line, which can be accessed by concurrent threads.

Building abstract threads. We still need to ensure the concurrent accesses of non-safety-critical TCB data are correct. For instance, we cannot allow a dead thread to

be put onto a ready thread queue. To address this issue, we build abstract threads to carry information of threads from TCBs to guide modifications by each other. In Fig. 3, we use the notation $[t]$ to specify the running thread, and the notation $\langle t \rangle$, for a ready thread. Here t is the identifier of the thread. With the knowledge about the existence of a ready thread B pointed by `next` (*i.e.*, $\langle B \rangle$), we know it is safe to switch to it via the operation `cswitch(A,next)`. Since abstract threads can be described in specifications, it allows us to write more intuitive and readable specifications for kernel code.

Treating abstract threads as resources. Like heap resources, abstract thread resources can be either local or shared. We can do *ownership transfers* on thread resources. When context switches, one thread will transfer some of the abstract thread resources (shared) along with the shared memory to the next thread. As shown in Fig. 3, when thread A context switches to thread B, the notation $[A]$ will be changed to $\langle A \rangle$ after context saving; $\langle A \rangle$ and $\langle B \rangle$ are transferred to the thread B along with the shared memory resource `next`; then $\langle B \rangle$ will be changed to $[B]$ after context loading. With transferred thread resources, thread B will know there is a ready thread A to switch to. Therefore, by treating abstract threads as resources, we find a simple and natural way to specify and reason about context switches. We design a proof system similar to CSL for modular verification with the support of ownership transfers on thread resources.

Defining concrete thread resources. To establish the soundness of our proof system, we must ensure that the abstract threads can be reified by concrete threads. The concrete representation of abstract threads, including stack, TCBs *etc*., can be defined globally. In Fig. 3, suppose that thread A is running, we ensure that there are two blocks of resources in the system. One of them is the running thread CThrd_A and the other is a ready thread RThrd_B. They correspond to the abstract threads $[A]$ and $\langle B \rangle$ in the assertions of thread A. We use the concrete thread resources to specify the global invariant of the machine, which allows us to prove the soundness of our proof system.

3 Machine model

In this section, we define a two-layer machine model. The physical machine we use is similar to realistic hardware, where no concept of thread exists. Based on it, we define an abstract machine with logical *abstract threads*, whose meta-data is abstracted into a thread pool. Moreover, the operation of context switch is abstracted as a primitive abstract instruction.

Physical machine. The formal definition of the physical machine is shown in Fig. 4 (left side). A machine configuration $\mathbb{W}$ consists of a code block $\mathbb{C}$, a memory block $\mathbb{M}$, a register file $\mathbb{R}$ and a program counter `pc`. The machine has 6 general registers. Some common instructions are defined to write programs in this paper. Their meanings, as well as the operational semantics, follow the conventions. For simplicity, we omit many realistic hardware details, *e.g.,* address alignment and bits-arithmetic.

Abstract machine. The abstract machine is shown in Fig. 4 (right side), where threads are introduced at this level. It is more intuitive to build a proof system (Sec. 4) to verify concurrent kernel code at this level. A thread pool P is a partial mapping from thread

$$
\begin{array}{rrcl}
\text{(PhyMach)} & \mathbb{W} & ::= & (\mathbb{C},\mathbb{M},\mathbb{R},\texttt{pc}) \\
\text{(PhyCode)} & \mathbb{C} & ::= & \{\texttt{f}:\texttt{i}\}^* \\
\text{(PhyMem)} & \mathbb{M} & ::= & \{\texttt{l}:\texttt{w}\}^* \quad (\texttt{l}=4n) \\
\text{(PhyRegFile)} & \mathbb{R} & ::= & \{\texttt{r}:\texttt{w}\}^* \\
\text{(Register)} & \texttt{r} & ::= & \texttt{v0} \mid \texttt{a0} \mid \texttt{a1} \mid \texttt{a2} \mid \texttt{sp} \mid \texttt{ra} \\
\text{(Instruction)} & \texttt{i} & ::= & \texttt{add}\ \texttt{r}_d,\ \texttt{r}_s \mid \texttt{addi}\ \texttt{r}_d,\ \texttt{w} \\
 & & & \mid \texttt{mov}\ \texttt{r}_d,\ \texttt{r}_s \mid \texttt{movi}\ \texttt{r}_d,\ \texttt{w} \\
 & & & \mid \texttt{lw}\ \texttt{r}_t,\ \texttt{w}(\texttt{r}_s) \mid \texttt{sw}\ \texttt{r}_t,\ \texttt{w}(\texttt{r}_s) \\
 & & & \mid \texttt{jmp}\ \texttt{f} \mid \texttt{call}\ \texttt{f} \mid \texttt{ret} \\
 & & & \mid \texttt{subi}\ \texttt{r}_d,\ \texttt{w} \mid \texttt{bz}\ \texttt{r}_t,\ \texttt{f}
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(AbsMach)} & W & ::= & (C,S,\texttt{pc}) \\
\text{(State)} & S & ::= & (M,R,P) \\
\text{(AbsCode)} & C & ::= & \{\texttt{f}:\texttt{c}\}^* \\
\text{(Mem)} & M & ::= & \{\texttt{l}:\texttt{w}\}^* \\
\text{(RegFile)} & R & ::= & \{\texttt{r}:\texttt{w}\}^* \\
\text{(TID)} & t & ::= & \texttt{w} \\
\text{(Pool)} & P & ::= & \{t:T\}^* \\
\text{(Thrd)} & T & ::= & \mathit{run} \mid (\mathit{rdy},R) \\
\text{(AbsInstr)} & \texttt{c} & ::= & \texttt{cswitch} \mid \texttt{i} \\
\text{(TIDList)} & L & ::= & t :: L \mid \mathsf{nil}
\end{array}
$$

Fig. 4. Physical and abstract machine models

IDs t to abstract threads T. Each abstract thread has a tag specifying its status, which is either running (*run*) or ready (*rdy*). Each ready thread has a copy of saved register file as its machine context data. The abstract instructions include an abstract operation of context switch (`cswitch`) and other physical machine instructions defined on the left. We model the operational semantics using the step transition relation $W \longmapsto W'$ defined in Fig. 5. The abstract instruction `cswitch` requires two thread IDs passed as arguments in `a0` and `a1`, one of which is tagged by *run* and the other is taged by *rdy* in the thread pool. After `cswitch`, the two abstract threads exchange tags, and the control of processor is passed from the old thread to the new one. The registers of old thread are saved in the source abstract thread and the registers in the destination thread are loaded into machine state. Except for `cswitch`, the state transitions of other instructions are similar to those of the physical machine.

Machine translation. In our proof system, once a program is proved safe at the abstract machine level, it should be proved safe as well at the physical machine level. We define a relation between abstract machine with physical machine (in the TR). The code block at the abstract machine level is extended with the code of implementation of context switch, and the abstract instruction `cswitch` is translated to a call instruction that invokes the implementation code of context switch. The memory block at the abstract machine level is translated to physical memory block by being merged with the memory where context data is stored. By the translation, it can be proved that any safe program over the abstract machine is safe over the physical machine.

4 Proof system

In this section, we extend the assertion language of CSL to specify the thread resources, and propose a small proof system supporting verification of concurrent code with modification of TCBs at the assembly level.

$((M,R,P),\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} ((M',R',P'),\mathtt{pc}')$	
if $\mathtt{c} =$	then
$\mathtt{i}$	$((M,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((M',R'),\mathtt{pc}') \wedge P = P'$
$\mathtt{cswitch}$	$\exists R'', P''.\, M = M' \wedge R'' = R\{\mathtt{ra} : \mathtt{pc}+1\} \wedge t = R(\mathtt{a0})$ $\wedge t' = R(\mathtt{a1}) \wedge \mathtt{pc}' = R'(\mathtt{ra})$ $\wedge P = \{t : run,\ \ t' : (rdy, R')\} \uplus P''$ $\wedge P' = \{t : (rdy, R''),\ \ t' : run\} \uplus P''$ R and R' is complete.

$((M,R),\mathtt{pc}) \overset{\mathtt{i}}{\hookrightarrow} ((M',R'),\mathtt{pc}')$	
if $\mathtt{i} =$	then
$\mathtt{add}\ \mathtt{r}_d, \mathtt{r}_s$	$M' = M \wedge R' = R\{\mathtt{r}_d : R(\mathtt{r}_d) + R(\mathtt{r}_s)\} \wedge \mathtt{pc}' = \mathtt{pc}+1$
$\mathtt{call}\ \mathtt{f}$	$M' = M \wedge R' = R\{\mathtt{ra} : \mathtt{pc}+1\} \wedge \mathtt{pc}' = \mathtt{f}$
$\mathtt{jmp}\ \mathtt{f}$	$M' = M \wedge R' = R \wedge \mathtt{pc}' = \mathtt{f}$
$\mathtt{ret}$	$M' = M \wedge R' = R \wedge \mathtt{pc}' = R(\mathtt{ra})$

$$\frac{C(\mathtt{pc}) = \mathtt{c} \qquad (S,\mathtt{pc}) \overset{\mathtt{c}}{\hookrightarrow} (S',\mathtt{pc}')}{(C,S,\mathtt{pc}) \longmapsto (C,S',\mathtt{pc}')}$$

Fig. 5. Operational semantics of abstract machine

4.1 Assertion language and code specification

We use p and q as assertion variables, which are predicates over machine states. The assertion constructs, adapted from separation logic [17], are *shallowly embedded* in the meta language , as shown in Fig. 6. In our assertion language, there are two special assertion constructs for abstract threads. One of them is $\langle t \rangle$ specifying a ready thread and the other is $[t]$ specifying a current running thread. Since threads are explicit resources in the abstract machine, their machine context data (values in registers) are preserved across context switch. Hence the resources of registers shouldn't be shared. We explicitly mark a pure assertion by $\sharp$, which forbids an assertion specifying resources. An unary notation ($\diamond p$) mark an assertion p that only specifies shared resources but no thread local resources (*e.g.,* registers). Registers are also treated as resources, and $\mathtt{r} \mapsto \mathtt{w}$ specifies a register with the value of $\mathtt{w}$. The notation $\mathtt{r}_1,\ldots,\mathtt{r}_n \mapsto \mathtt{w}_1,\ldots,\mathtt{w}_n$ is a compact form for multiple registers.

We borrow the idea from SCAP [4] and use a (p,g) pair to specify instructions at assembly-level. The pre-condition p describes the state before the first instruction of an instruction sequence, while the action g describes the actions done by the whole instruction sequence. In the proof system, each instruction is associated with a (p,g) pair, where g describes the actions from this instruction to the end of the current function. For all instructions in C, their (p,g) pairs are put in Ψ, a global mapping from labels to specifications. The specification form (p,g) is different from the traditional pre-condition and post-condition, which are both assertions and related by auxiliary variables. We can still use a notation to specify instructions in the traditional style,

$$
\begin{array}{rcl}
\mathsf{true} & \triangleq & \lambda(M,R,P).\ \mathsf{True} \\
\mathsf{false} & \triangleq & \lambda(M,R,P).\ \mathsf{False} \\
\mathsf{emp} & \triangleq & \lambda(M,R,P).M=\{\cdot\} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
p * q & \triangleq & \lambda(M,R,P).\exists M_1,M_2,R_1,R_2,P_1,P_2.M=M_1 \uplus M_2 \wedge R=R_1 \uplus R_2 \wedge P=P_1 \uplus P_2 \\
 & & \quad \wedge p\ (M_1,R_1,P_1) \wedge q\ (M_2,R_2,P_2) \\
p -\!\!* q & \triangleq & \lambda(M,R,P).\forall M_1,R_1,P_1,M',R',P'.(M'=M_1 \uplus M \wedge R'=R_1 \uplus R \wedge P'=P_1 \uplus P) \\
 & & \quad \rightarrow p\ (M_1,R_1,P_1) \rightarrow q\ (M',R',P') \\
p \mathbb{M} q & \triangleq & \lambda S.(p\ S) \wedge (q\ S) \\
p \mathbb{W} q & \triangleq & \lambda S.(p\ S) \vee (q\ S) \\
\exists v.p & \triangleq & \lambda S.\exists v.p\ S \\
\sharp p & \triangleq & \lambda(M,R,P).p \wedge M=\{\cdot\} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
\diamond p & \triangleq & \lambda(M,R,P).p\ (M,R,P) \wedge R=\{\cdot\} \\
\mathtt{r} \mapsto \mathtt{w} & \triangleq & \lambda(M,R,P).R=\{\mathtt{r}:\mathtt{w}\} \wedge M=\{\cdot\} \wedge P=\{\cdot\} \\
\mathtt{r} \hookrightarrow \mathtt{w} & \triangleq & \lambda(M,R,P).\exists R'.R=\{\mathtt{r}:\mathtt{w}\} \uplus R' \\
\mathtt{l} \mapsto \mathtt{w} & \triangleq & \lambda(M,R,P).M=\{\mathtt{l}:\mathtt{w}\} \wedge \mathtt{l} \neq \mathsf{NULL} \wedge R=\{\cdot\} \wedge P=\{\cdot\} \\
[t] & \triangleq & \lambda(M,R,P).P=\{t:run\} \wedge t \neq \mathsf{NULL} \wedge M=\{\cdot\} \wedge R=\{\cdot\} \\
\langle t \rangle & \triangleq & \lambda(M,R,P).P=\{t:(rdy,\_)\} \wedge t \neq \mathsf{NULL} \wedge M=\{\cdot\} \wedge R=\{\cdot\}
\end{array}
$$

Fig. 6. Definition of selected assertion constructs

$$
\left\{ \begin{array}{l} p \\ q \end{array} \right\}^{(v_1,\ldots,v_n)} \triangleq (\lambda S.\exists v_1,\ldots,v_n.(p(v_1,\ldots,v_n) * \mathsf{true})\ S, \\
\qquad \lambda S,S'.\forall p'.\forall v_1,\ldots,v_n.(p(v_1,\ldots,v_n) * p')\ S \rightarrow (q(v_1,\ldots,v_n) * p')\ S')
$$

where p is the pre-condition of instructions, q is the post-condition, and $v_1,\ldots,v_n$ are auxiliary variables occurring in the precondition and the postcondition. We define a binary operator for composing two pairs into one.

$$
(p,g) \triangleright (p',g') \triangleq (\lambda S.p\ S \wedge (\forall S'.g\ S\ S' \rightarrow p'\ S'), \\
\qquad \lambda S,S''.p\ S \rightarrow (\exists S'.g\ S\ S' \wedge g'\ S'\ S''))
$$

If an instruction sequence satisfies (p,g) and the following instruction sequence satisfies (p',g'), then the composed instruction sequence would satisfy $(p,g) \triangleright (p',g')$. The weakening relation between two pairs is defined as below:

$$
(p,g) \Rightarrow (p',g') \triangleq \forall S.p\ S \rightarrow p'\ S \wedge (\forall S'.g'\ S\ S' \rightarrow g\ S\ S')
$$

i.e., the precondition p be stronger than p' and the action g be weaker than g'.

$$
\begin{array}{llcl}
\text{(Assert)} & p,q & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{emp} \mid p * q \mid p -\!\!* q \mid p \mathbb{M} q \mid p \mathbb{W} q \mid \exists v.p \mid \mathtt{l} \mapsto \mathtt{w} \\
 & & & \mid [t] \mid \langle t \rangle \mid \sharp p \mid \diamond p \mid \mathtt{r} \mapsto \mathtt{w} \mid \mathtt{r} \hookrightarrow \mathtt{w} \\
\text{(Action)} & g & \in & \mathsf{State} \rightarrow \mathsf{State} \rightarrow \mathsf{Prop} \\
\text{(Spec)} & \Psi & ::= & \{\mathtt{f}:(p,g)\}^*
\end{array}
$$

4.2 Invariant for shared resources and inference rules

As mentioned previously, our proof system draws ideas of ownership transfer from CSL. By defining invariants for shared resources, our proof system ensures safe operations of TCBs.

Unlike the invariant in concurrent separation logic, the invariant of shared resources defined in our proof system is parameterized by two thread IDs: $I(t_s, t_d)$. Briefly, the invariant describes the shared resources before context switch with the direction from the thread t_s to t_d. One of the benefits of parameters is that the invariant is thread-specific.

Like the abstract invariant I in CSL, the invariant $I(t_s, t_d)$ is abstract and can be instantiated to concrete definitions to verify various programs, as long as the instantiation satisfies the requirement of being *precise* [17].

Precisely, the invariant $I(t_s, t_d)$ describes the shared resources when the context switch is invoked from the thread t_s to the thread t_d, but *excluding the resources of the two threads*. Since the control flow from one thread to another is *deterministic* by context switch, every two threads may negotiate a particular invariant that is different from pairs of other threads. We can define different assertions (of shared resources) which depend on the source and the destination threads of a context switch. This is quite different from concurrent code at user-level, where a context switch is non-deterministic and the scheduling algorithm is abstracted away.

The judgment for instructions in our proof system is of the following form: $\Psi, I \vdash \{(p,g)\}\ \texttt{pc} : \texttt{c}$, where Ψ and I are given as specifications. The judgement states that an instruction sequence, started with `c` at the label of `pc` and ended with a `ret`, satisfies specification (p,g) under Ψ and I. Some selected inference rules for instructions are shown in Fig. 7.

In the rule of (ADD), the premise says that the specification (p,g) implies the action of the `add` instruction composed with the specification of the next instruction, $\Psi(\texttt{pc}+1)$. The action of `add` instruction is that if the destination register $\texttt{r}_d$ contains the value of w_1, and the source register $\texttt{r}_s$ contains the value of w_2, then after the instruction, $\texttt{r}_d$ will contain the sum of w_1 and w_2, while $\texttt{r}_s$ will remain unchanged.

Functions are reasoned with the rules of (CALL) and (RET). The (CALL) rule says that the specification (p,g) implies the action that is composed by (1) the action of instruction `call`, (2) the specification of the *function* invoked $\Psi(\texttt{f})$, (3) the action of instruction `ret`, and (4) the specification of the next instruction $\Psi(\texttt{pc}+1)$. The (RET) rule says that the specification (p,g) implies an empty action, which means the actions of the current function should be fulfilled.

The most important rule is (CSW). The precondition of `cswitch` requires the following resources: the current thread resource, the registers `a0` containing the current thread ID t and `a1` containing the destination thread ID t', and the shared resource satisfying the invariant $\diamond I(t,t')$. After return from context switch, the current thread will own the shared resources (satisfying $\diamond I(t'',t)$ for some t'') again.

4.3 Invariant of global resources and soundness

Each abstract thread corresponds to the part of global resources representing the concrete resources allocated for this thread. For example, for an abstract thread $\langle t \rangle$, there

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}(\mathtt{r}_d \mapsto w1) * (\mathtt{r}_s \mapsto w2)\\ (\mathtt{r}_d \mapsto w1{+}w2) * (\mathtt{r}_s \mapsto w_2)\end{array}\right\}^{(w1,w2)} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{add}\ \mathtt{r}_d,\ \mathtt{r}_s}\ (\textsc{add})$$

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}\mathtt{ra} \mapsto \_\\ \mathtt{ra} \mapsto \mathtt{pc}{+}1\end{array}\right\} \triangleright \Psi(\mathtt{f}) \triangleright \left\{\begin{array}{l}\mathtt{ra} \mapsto \mathtt{pc}{+}1\\ \mathtt{ra} \mapsto \_\end{array}\right\} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{call}\ \mathtt{f}}\ (\textsc{call})$$

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}\mathsf{emp}\\ \mathsf{emp}\end{array}\right\}}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{ret}}\ (\textsc{ret}) \qquad \frac{(p,g) \Rightarrow \Psi(\mathtt{f})}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{jmp}\ \mathtt{f}}\ (\textsc{jmp})$$

$$\frac{(p,g) \Rightarrow \left\{\begin{array}{l}[t] * (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto t, t', \_) * \langle t' \rangle * \diamond I(t,t')\\ [t] * (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto t, t', \_) * \exists t''.\, \langle t'' \rangle * \diamond I(t'',t)\end{array}\right\}^{(t,t')} \triangleright \Psi(\mathtt{pc}{+}1)}{\Psi, I \vdash \{(p,g)\}\ \mathtt{pc} : \mathtt{cswitch}}\ (\textsc{csw})$$

Fig. 7. Inference rules (selected)

exist resources of its TCB, stack, and private resources. Therefore, all resources can be divided into parts and each of them is associated to one thread. The global invariant GINV, defined in Fig. 8, describes the partition of all resources globally. The invariant is the key for proving the soundness theorem of our proof system.

First, for each thread, we define a predicate Cont to specify its resources and control flow, i.e. the *continuation* of this thread. The first parameter n of this predicate specifies the number of functions nested in the thread's control flow. If n is equal to zero, it means that the thread is running in the topmost function, which is required to be an infinite loop and cannot return. If the number n is greater than zero, the predicate says that there is a specification (p,g) in Ψ at pc, such that the resources of the thread satisfies p; and g guarantees that the thread will continue to satisfy Cont recursively after it returns to the address *retaddr*.

The concrete resources of a *running thread* are specified by a continuation Cont with an additional condition, the running thread owns all registers. The parameter pc points to the next instruction the thread is going to run. Here we use an abbreviation $\lfloor R \rfloor$ to denote the resources of all registers, except that the value in ra is of no interest.

For a *ready thread* (or a runnable thread), its concrete resources are defined by separating implication $-\!\!*$: if given (1) the resources of saved machine context $\lfloor R \rfloor$, (2) the abstract resource of itself $[t]$, (3) another ready thread t' and (4) shared resources specified by $\diamond I(t',t)$, the resources of the ready thread can be transformed into the resources of a running thread. Its thread ID is specified by the second parameter of RThrd, and the third parameter is the machine context data saved in its TCB. Please note that the program counter of a ready thread is saved into the register ra.

The whole machine state can be partitioned, and each part is owned by one thread, which is either running or ready. Thus, the global invariant GINV is defined in the form of separating conjunction by CThrd and RThrd. The structure of GINV is isomorphic to the thread pool P: the abstract running thread is mapped to the resource specified by

$$
\begin{array}{rcl}
\lfloor R \rfloor & \triangleq & (\mathtt{ra} \mapsto \_) * (\mathtt{v0} \mapsto R(\mathtt{v0})) * (\mathtt{sp} \mapsto R(\mathtt{sp})) \\
& & \quad * (\mathtt{a0} \mapsto R(\mathtt{a0})) * (\mathtt{a1} \mapsto R(\mathtt{a1})) * (\mathtt{a2} \mapsto R(\mathtt{a2})) \\
\mathsf{Cont}(n{+}1, \Psi, \mathtt{pc}) & \triangleq & \lambda S.\, \Psi(\mathtt{pc}) = (p, g) \wedge (p\ S) \\
& & \wedge (\forall S'.\, g\ S\ S' \rightarrow (\exists\, \mathit{retaddr}.\, (\mathtt{ra} \hookrightarrow \mathit{retaddr}) \mathbin{-\!\!\!\circledast} \mathsf{Cont}(n, \Psi, \mathit{retaddr}))\ S') \\
\mathsf{Cont}(0, \Psi, \mathtt{pc}) & \triangleq & \lambda S.\, \Psi(\mathtt{pc}) = (p, g) \wedge (p\ S) \wedge (\forall S'.\, g\ S\ S' \rightarrow \mathsf{False}) \\
\mathsf{CThrd}(\Psi, t, \mathtt{pc}) & \triangleq & \exists n.\, \mathsf{Cont}(n, \Psi, \mathtt{pc}) \mathbin{-\!\!\!\circledast} ([t] * \exists R.\, \lfloor R \rfloor * \mathsf{true}) \\
\mathsf{RThrd}(\Psi, t, R) & \triangleq & \lfloor R \rfloor * [t] * \exists t'.\, \langle t' \rangle * \diamond I(t', t) \mathrel{-\!\!*} \mathsf{CThrd}(\Psi, t, R(\mathtt{ra})) \\
\mathsf{GINV}(\Psi, P, \mathtt{pc}) & \triangleq & \mathsf{CThrd}(\Psi, t, \mathtt{pc}) * \mathsf{RThrd}(\Psi, t_0, R_0) * \cdots * \mathsf{RThrd}(\Psi, t_n, R_n) \\
& & \text{where } P = \{t : \mathit{run},\ t_0 : (\mathit{rdy}, R_0),\ \ldots,\ t_n : (\mathit{rdy}, R_n)\}
\end{array}
$$

Fig. 8. Concrete threads and the global invariant

```
struct tcb {           | void schedule_p2()
  struct context ctxt; |  {
  struct tcb *prev;    |    struct tcb *old, *new;
  struct tcb *next;    |    old = cur;
};                     |    new = deq(&rq);
struct queue {         |    if (new == NULL) return;
  struct tcb *head;    |    enq(&rq, old);
  struct tcb *tail;    |    cur = new;
};                     |    cswitch(old,new);
struct tcb *cur;       |    return;
struct queue rq;       | }
```

Fig. 9. Pseudo C code for `schedule_p2()`

CThrd; an abstract ready thread is mapped to a resource specified by RThrd. Note that GINV requires that there be one and only one running abstract thread, since the physical machine has only one single processor. Our proof system ensures that the machine state always satisfies the global invariant, ($\mathsf{GINV}(\Psi, P, \mathtt{pc})\ (M, R, P)$).

The soundness property of our proof system states that any program that is well-formed in our proof system will run safely on the abstract machine. The property can be proved by the global invariant GINV, which always holds through machine execution. We can first prove that if every machine configuration satisfies GINV, it can run forward for one step. And we can also prove that if a machine configuration (satisfying GINV) can proceed, the next machine configuration will also satisfy GINV. Hence by the invariant GINV, the soundness theorem of our proof system can be proved. The proof of the soundness theorem has been formalized in Coq [7].

5 Verification cases

In this section, we show how to use the proof system to verify two schedulers of pattern (II) and (III) shown in Fig. 1. We give the code written in pseudo C to explain

the programs and their specifications. The corresponding assembly code and selected assertions of the two schedulers are shown in Fig. 10.

Scheduler as function. The scheduler function `schedule_p2()` (see Fig. 9) follows the process discussed in Sec. 2. The functions `deq()` and `enq()` are used to remove and insert nodes in thread queues. The main task of the scheduler is to choose a candidate from the thread queue and then perform context switch from the current thread to the candidate. There are two global variables, `cur` and `rq`. The variable `cur` points to the TCB of the running thread; `rq` points to the thread queue containing TCBs of all other runnable (ready) threads.

The notation $t \overset{field}{\longmapsto} w$ specifies a named field in the structure. The notation $\mathsf{ptcb}(t)$ specifies a part of TCB including the fields of `next` and `prev`. The predicate $\mathsf{RQ}(q,L)$ specifies a doubly linked list as a thread queue pointed to by q, where L is a list of thread IDs of the thread queue. We also use $\langle L\rangle$ as an abbreviation for $\langle t_0\rangle * \langle t_1\rangle * \cdots * \langle t_n\rangle$, if L is $t_0 :: t_1 :: \cdots :: t_n :: \mathsf{nil}$, and use $\mathtt{l} \mapsto {}^{(n)}\_$ to specify n continuous memory cells.

$$
\begin{array}{rcl}
\mathtt{l} \overset{field}{\longmapsto} w & \triangleq & (l+\textit{offset of the field in the struct}) \mapsto w \\
\mathsf{ptcb}(t) & \triangleq & (t \overset{prev}{\longmapsto} \_) * (t \overset{next}{\longmapsto} \_) \\
\mathsf{RQseg}(pv,tl,t,\mathsf{nil}) & \triangleq & (t \overset{prev}{\longmapsto} pv) * \exists t'.\,(t \overset{next}{\longmapsto} \mathsf{NULL}) * \sharp(t=tl) \\
\mathsf{RQseg}(pv,tl,t,t' :: L') & \triangleq & (t \overset{prev}{\longmapsto} pv) * (t \overset{next}{\longmapsto} t') * \mathsf{RQseg}(t,tl,t',L') \\
\mathsf{RQ}(q,\mathsf{nil}) & \triangleq & (q \overset{head}{\longmapsto} \mathsf{NULL}) * (q \overset{tail}{\longmapsto} \mathsf{NULL}) \\
\mathsf{RQ}(q,t :: L) & \triangleq & \exists pv.\exists tl.\,(q \overset{head}{\longmapsto} t) * (q \overset{tail}{\longmapsto} tl) * \mathsf{RQseg}(pv,tl,t,L) \\
\mathsf{K}(bp,n,\mathtt{w}_0 :: \mathtt{w}_1 :: \ldots :: \mathtt{w}_m :: \mathsf{nil}) & \triangleq & \exists sp.\,(\mathtt{sp} \mapsto sp) * \sharp(sp=bp+4n) * (bp \mapsto {}^{(n)}\_) \\
 & & *(sp \mapsto \mathtt{w}_0) * (sp+4 \mapsto \mathtt{w}_1) * \cdots * (sp+4m \mapsto \mathtt{w}_m) \\
\mathsf{K}(bp,n) & \triangleq & \mathsf{K}(bp,n,\mathsf{nil})
\end{array}
$$

The specification of `schedule_p2()` is shown below:

$$
\left\{
\begin{array}{l}
[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L.\,\mathsf{RQ}(\mathtt{rq},L) * \langle L\rangle * (\mathtt{ra} \mapsto ret) \\
\qquad\qquad * \mathsf{K}(bp,20) * (\mathtt{v0},\mathtt{a0},\mathtt{a1} \mapsto \_,\_,\_) \\
[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L.\,\mathsf{RQ}(\mathtt{rq},L) * \langle L\rangle * (\mathtt{ra} \mapsto ret) \\
\qquad\qquad * \mathsf{K}(bp,20) * (\mathtt{v0},\mathtt{a0},\mathtt{a1} \mapsto \_,\_,\_)
\end{array}
\right\}^{(t,ret,bp)}
$$

Here we use a notation $\mathsf{K}(bp,n,w :: w' :: \cdots)$ to describe a stack frame. The first parameter bp is the base address of a stack frame. The second parameter n is the size of unused space (number of words). And the third parameter is a list of words, representing the values on stack top down, that is, the leftmost value in the list is the topmost value in the stack frame. If the stack frame is empty, we omit the third parameter.

The abstract invariant I is instantiated to a concrete definition specifying the shared resources *before* and *after* context switch for this implementation of scheduler.

$$
I(t,t') \triangleq \mathsf{ptcb}(t') * (\mathtt{cur} \mapsto t') * \exists L.\,\mathsf{RQ}(\mathtt{rq},t :: L) * \langle L\rangle
$$

schedule_p2:

$\{[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L.\,\mathsf{RQ}(\mathtt{rq}, L)$
$* \langle L \rangle * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, \mathit{ret})$
$* \mathsf{K}(\mathit{bp}, 20)\}$

```
subi    sp,   12
sw      ra,   8(sp)
movi    a0,   cur
lw      v0,   0(a0)
sw      v0,   0(sp)
```

$\{[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L.\,\mathsf{RQ}(\mathtt{rq}, L)$
$* \langle L \rangle * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{cur}, \_, t, \_)$
$* \mathsf{K}(\mathit{bp}, 17, t :: \_ :: \mathit{ret} :: \mathsf{nil})\}$

```
movi    a0,   rq
call    deq
bz      v0,   Ls_ret
```

$\{[t] * \mathsf{ptcb}(t) * \langle t' \rangle * \mathsf{ptcb}(t') * \exists L.\,\mathsf{RQ}(\mathtt{rq}, L)$
$* \langle L \rangle * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{rq}, \_, t', \_)$
$* \mathsf{K}(\mathit{bp}, 17, t :: \_ :: \mathit{ret} :: \mathsf{nil}) * (\mathtt{cur} \mapsto t)\}$

```
sw      v0,   4(sp)
lw      a1,   0(sp)
call    enq
```

$\{[t] * \langle t' \rangle * \mathsf{ptcb}(t') * \exists L.\,\mathsf{RQ}(\mathtt{rq}, t :: L) * \langle L \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{rq}, t, 0, \_)$
$* \mathsf{K}(\mathit{bp}, 17, t :: t' :: \mathit{ret} :: \mathsf{nil}) * (\mathtt{cur} \mapsto t)\}$

```
lw      a1,   4(sp)
movi    a0,   cur
sw      a1,   0(a0)
lw      a0,   0(sp)
```

$\{[t] * \langle t' \rangle * \exists L.\,\mathsf{RQ}(\mathtt{rq}, t :: L) * \langle L \rangle * \mathsf{ptcb}(t')$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto t, t', 0, \_)$
$* \mathsf{K}(\mathit{bp}, 17, t :: t' :: \mathit{ret} :: \mathsf{nil}) * (\mathtt{cur} \mapsto t')\}$

```
cswitch
```

$\{[t] * \mathsf{ptcb}(t) * \exists t''.\,\langle t'' \rangle * \exists L.\,\mathsf{RQ}(\mathtt{rq}, t'' :: L)$
$* \langle L \rangle * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto t, t', \_, \_)$
$* \mathsf{K}(\mathit{bp}, 17, t :: \_ :: \mathit{ret} :: \mathsf{nil}) * (\mathtt{cur} \mapsto t)\}$

Ls_ret:

```
lw      ra,   8(sp)
addi    sp,   12
```

$\{[t] * \mathsf{ptcb}(t) * (\mathtt{cur} \mapsto t) * \exists L.\,\mathsf{RQ}(\mathtt{rq}, L)$
$* \langle L \rangle * (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, \mathit{ret})$
$* \mathsf{K}(\mathit{bp}, 20)\}$

```
ret
```

schedth:

$\{[\mathtt{sched}] * (\mathtt{cur} \mapsto \_) * \exists L.\,\mathsf{RQ}(\mathtt{rq}, L) * \langle L \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \_, \_, \_, \_)$
$* \exists \mathit{bp}.\,\mathsf{K}(\mathit{bp}, 10)\}$

```
movi    a0,   rq
call    deq
bz      v0,   schedth
movi    a2,   cur
sw      v0,   0(a2)
mov     a1,   v0
lw      a0,   sched
```

$\{[\mathtt{sched}] * \langle t' \rangle * (\mathtt{cur} \mapsto t') * \mathsf{ptcb}(t')$
$* \exists L.\,\mathsf{RQ}(\mathtt{rq}, L) * \langle L \rangle$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{sched}, t', \_, \_)$
$* \exists \mathit{bp}.\,\mathsf{K}(\mathit{bp}, 10)\}$

```
cswitch
```

$\{[\mathtt{sched}] * \exists t''.\,\langle t'' \rangle * \mathsf{ptcb}(t'') * (\mathtt{cur} \mapsto t'')$
$* \exists L.\,\mathsf{RQ}(\mathtt{rq}, L) * \langle L \rangle * \exists \mathit{bp}.\,\mathsf{K}(\mathit{bp}, 10)$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{v0}, \mathtt{ra} \mapsto \mathtt{sched}, \_, \_, \_)\}$

```
movi    a0,   rq
lw      a1,   0(a2)
call    enq
jmp     schedth
```

schedule_p3:

$\{[t] * \mathsf{ptcb}(t) * \langle \mathtt{sched} \rangle * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 10)\}$

```
subi    sp,   4
sw      ra,   0(sp)
movi    a1,   cur
lw      a0,   0(a1)
movi    a1,   sched
```

$\{[t] * \mathsf{ptcb}(t) * \langle \mathtt{sched} \rangle * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto t, \mathtt{sched}, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 9, \mathit{ret})\}$

```
cswitch
```

$\{[t] * \mathsf{ptcb}(t) * \langle \mathtt{sched} \rangle * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 9, \mathit{ret})\}$

```
lw      ra,   0(sp)
addi    sp,   4
```

$\{[t] * \mathsf{ptcb}(t) * \langle \mathtt{sched} \rangle * (\mathtt{cur} \mapsto t)$
$* (\mathtt{a0}, \mathtt{a1}, \mathtt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 10)\}$

```
ret
```

Fig. 10. Verification of `schedule_p2()`, `schedth()` and `schedule_p3()`

```
struct tcb sched;       |  schedth()
struct tcb *cur;        |  {
struct queue rq;        |    while(1){
schedule_p3()           |      cur = deq(&rq);
{                       |      cswitch(&sched, cur);
  cswitch(cur,&sched);  |      enq(&rq, cur);
  return;               |    }
}                       |  }
```

Fig. 11. Pseudo C code for `schedule_p3()`

Scheduler as a separated thread. A scheduler in the pattern (III) is implemented as a separated thread (see Fig. 11), which does scheduling jobs in an infinite loop. A global variable `sched` is added to represent the TCB of the scheduler thread. A stub function `schedule_p3()` can be invoked by other threads to do scheduling. As shown below, the specification of `schedule_p3()` function is different from the one of `schedule_p2()`. The schedule function in this implementation doesn't own the thread queue, which is owned by the scheduler thread $\langle \texttt{sched} \rangle$ instead since all of the operations over the thread queue are put into the separated thread.

$$\left\{ \begin{array}{l} [t] * \mathsf{ptcb}(t) * (\texttt{cur} \mapsto t) * \langle \texttt{sched} \rangle * (\texttt{a0}, \texttt{a1}, \texttt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 10) \\ [t] * \mathsf{ptcb}(t) * (\texttt{cur} \mapsto t) * \langle \texttt{sched} \rangle * (\texttt{a0}, \texttt{a1}, \texttt{ra} \mapsto \_, \_, \mathit{ret}) * \mathsf{K}(\mathit{bp}, 10) \end{array} \right\}^{(t, \mathit{bp}, \mathit{ret})}$$

The specification of `schedth()` function is shown below:

$$\left\{ \begin{array}{l} [\texttt{sched}] * (\texttt{cur} \mapsto \_) * \exists L \,.\, \mathsf{RQ}(\texttt{rq}, L) * \langle L \rangle \\ \qquad\qquad * (\texttt{a0}, \texttt{a1}, \texttt{a2}, \texttt{v0}, \texttt{ra} \mapsto \_, \_, \_, \_, \_) * \exists \mathit{bp} \,.\, \mathsf{K}(\mathit{bp}, 10) \\ \mathsf{false} \end{array} \right\}$$

Since the ready thread queue is only owned by the scheduler thread, it does not need to be shared by other threads and occur in the invariant for the shared resources, I:

$$I(t, t') \triangleq (\sharp(t' = \texttt{sched}) * (\texttt{cur} \mapsto t) * \mathsf{ptcb}(t)) \mathbb{W} (\sharp(t = \texttt{sched}) * (\texttt{cur} \mapsto t') * \mathsf{ptcb}(t'))$$

The invariant $I(t, t')$ is defined by two cases on the direction of context switch: if the destination thread is the scheduler thread, $I(t, t')$ requires that the value in `cur` be equal to the ID of the source thread, t; or if the source thread is the scheduler thread, $I(t, t')$ requires that the value in `cur` be equal to the ID of the destination thread.

6 Related work and conclusions

Gotsman and Yang [6] proposed a two-layer framework to verify schedulers. The proof system in the lower-layer is for verifying code manipulating TCBs, while the upper-layer is for verifying the rest concurrent code of the kernel. Since thread queues and TCBs are hidden from the upper-layer, one thread could not have any knowledge of the others, thus their proof system is unable to verify the scheduling pattern of II and III. Similar to our assertion $\mathsf{RThrd}(\cdots)$, they introduced a primitive predicate *Process*(G) to

relate TCBs in the lower-layer with threads in the upper-layer, but there is no counterpart of $\langle t \rangle$ in their framework.

Feng *et al.* also verified a kernel prototype [3] in a two-layer framework. Code manipulating TCBs needs to be verified in the lower-layer of their framework. The TCBs are connected with actual threads in the upper layer by an interpretation function of their framework. Our use of global invariant is similar to their use of the interpretation function. In the upper-layer, information of threads is completely hidden. Thus, their framework also fails to support the verification of the scheduler pattern of II and III.

Ni *et al.* verified a small thread manager with a logic system [15,14] supporting modular reasoning about code including embedded code pointers. In their logic, however, there is no abstraction of threads. Multithreaded programs are seen as sequential interleaving of pieces of code in low-level continuation passing style. Therefore, TCBs with embedded code pointers can be treated as normal data. But since the reasoning level is too low without any abstraction, TCBs have to be specified by over-complicated logic expressions and then it is very difficult to apply their method to realistic code.

Klein *et al.* verified a micro-kernel, seL4 [11], where the kernel code runs sequentially. Thus they used a sequential proof system to verify most of the kernel code. The scheduling pattern of seL4 is similar to our pattern I, but they trusted the code doing context saving and loading, and left it unverified. Since they do not verify user processes upon the kernel, they need not relate TCBs in the kernel with actual user processes.

Gargano *et al.* used a framework CVM [5] to build verified kernels in the Verisoft project. CVM is a computational model for concurrent user processes, which interleave through a micro-kernel. Starostin and Tsyban presented a formal approach [18] to reason about context switch between user processes. The context switch code and proofs are integrated in a framework for building verified kernels (CVM) [10]. Their framework keeps a global invariant, *weak consistency*, to relate TCBs in the kernel with user processes outside the kernel. Since the kernel itself is sequential, their process scheduling follows pattern I. The other two patterns cannot be verified.

In this paper, we proposed a novel approach to verify concurrent thread management code, which allows multiple threads to modify their own thread control blocks. The assertions of the code and inference rules of the proof system are straightforward and easy to follow. Moreover, it can be easily extended to support other kernel features (e.g., preemptive scheduling, multi-core systems, synchronizations) and to be practically applied to realistic OS code.

Acknowledgements. We thank anonymous referees for suggestions and comments on an earlier version of this paper. Yu Guo, Xinyu Feng and Peizhi Shi are supported in part by grants from National Natural Science Foundation of China (Nos. 61073040, 61202052 and 61229201), the Fundamental Research Funds for the Central Universities (Nos. WK0110000018 and WK0110000025), and Program for New Century Excellent Talents in Universities (NCET). Zhong Shao is supported in part by DARPA under agreement numbers FA8750-10-2-0254 and FA8750-12-2-0293, and by NSF grants CNS-0910670, CNS-0915888, and CNS-1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. R. S. Engelschall. Portable multithreading: the signal stack trick for user-space thread creation. In *Proc. of ATEC'00*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.
2. D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.
3. X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *Proc. VSTTE'08*, pages 54–69, Toronto, Canada, October 2008.
4. X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *Proc. PLDI'06*, pages 401–414, June 2006.
5. M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *Proc. TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.
6. A. Gotsman and H. Yang. Modular verification of preemptive os kernels. In *Proc. ICFP'11*, pages 404–417, Tokyo, Japan, 2011. ACM.
7. Y. Guo, X. Feng, Z. Shao, and P. Shi. Modular verification of concurrent thread management (technical report and coq proof). `http://kyhcs.ustcsz.edu.cn/~guoyu/sched/`, June 2012.
8. J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40:80–89, July 2006.
9. M. Hohmuth and H. Tews. The vfiasco approach for a verified operating system. In *Proceedings of the 2nd ECOOP Workshop on Programming Languages and Operating Systems*, 2005.
10. T. In der Rieden and A. Tsyban. CVM – A verified framework for microkernel programmers. In *Proc. SSV'08*, volume 217C of *Electronic Notes in Theoretical Computer Science*, pages 151–168. Elsevier Science B.V., 2008.
11. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.
12. R. Love. *Linux Kernel Development (2nd Edition) (Novell Press)*. Novell Press, 2005.
13. M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.
14. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. POPL'06*, pages 320–333, Jan. 2006.
15. Z. Ni, D. Yu, and Z. Shao. Using XCAP to certify realistic systems code: Machine context management. In *Proc. TPHOLs'07*, volume 4732 of *Lecture Notes in Computer Science*, pages 189–206. Springer-Verlag, September 2007.
16. P. W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
18. A. Starostin and A. Tsyban. Verified process-context switch for C-programmed kernels. In J. Woodcock and N. Shankar, editors, *Proc. VSTTE'08*, volume 5295 of *Lecture Notes in Computer Science*, pages 240–254, Toronto, Canada, Oct. 2008. Springer.

Compositional Verification of a Baby Virtual Memory Manager

Alexander Vaynberg and Zhong Shao

Yale University

Abstract. A virtual memory manager (VMM) is a part of an operating system that provides the rest of the kernel with an abstract model of memory. Although small in size, it involves complicated and interdependent invariants that make monolithic verification of the VMM and the kernel running on top of it difficult. In this paper, we make the observation that a VMM is constructed in layers: physical page allocation, page table drivers, address space API, etc., each layer providing an abstraction that the next layer utilizes. We use this layering to simplify the verification of individual modules of VMM and then to link them together by composing a series of small refinements. The compositional verification also supports function calls from less abstract layers into more abstract ones, allowing us to simplify the verification of initialization functions as well. To facilitate such compositional verification, we develop a framework that assists in creation of verification systems for each layer and refinements between the layers. Using this framework, we have produced a certification of BabyVMM, a small VMM designed for simplified hardware. The same proof also shows that a certified kernel using BabyVMM's virtual memory abstraction can be refined following a similar sequence of refinements, and can then be safely linked with BabyVMM. Both the verification framework and the entire certification of BabyVMM have been mechanized in the Coq Proof Assistant.

1 Introduction

Software systems are complex feats of engineering. What makes them possible is the ability to isolate and abstract modules of the system. In this paper, we consider an operating system kernel that uses virtual memory. The majority of the kernel makes an assumption that the memory is a large space with virtual addresses and a specific interface that allows the kernel to request access to any particular page in this large space. In reality, this entire model of memory is in the imagination of the programmer, supported by a relatively small but important portion of the kernel called the virtual memory manager. The job of the virtual memory manager is to handle all the complexities of the real machine architecture to provide the primitives that the rest of the kernel can use. This is exactly how the programmer would reason about this software system.

However, when we consider verification of such code, current approaches are mostly monolithic in nature. Abstraction is generally limited to abstract data types, but such abstraction can not capture changes in the semantics of computation. For example, it is impossible to use abstract data types to make virtual memory appear to work like physical memory without changing operational semantics. To create such abstraction, a

change of computational model is required. In the Verisoft project[11, 18], the abstract virtual memory is defined by creating the CVM model from VAMP architecture. In AIM[7], multiple machines are used to define interrupts in the presence of a scheduler.

These transitions to more abstract models of computation tend to be quite rare, and when present tend to be complex. The previously mentioned VAMP-CVM jump in Verisoft abstracts most of kernel functionality in one step. In our opinion, it would be better to have more abstract computation models, with smaller jumps in abstraction. First, it is easier to verify code in the most abstract computational model possible. Second, smaller abstractions tend to be easier to prove and to maintain, while larger abstractions can be still achieved by composing the smaller ones. Third, more abstractions means more modularity; changes in the internals of one module will not have global effects.

However, we do not commonly see Hoare-logic verification that encourages multiple models. The likely reason is that creating abstract models and linking across them is seen as ad-hoc and tedious additional work. In this paper we show how to reduce the effort required to define models and linking, so that code verification using multiple abstractions becomes an effective approach. More precisely, our paper makes the following contributions:

- We present a framework for quickly defining multiple abstract computational models and their verification systems.
- We show how our framework can be used to define safe cross-abstraction linking.
- We show how to modularize a virtual memory manager and define abstract computational models for each layer of VMM.
- We show a complete verification of a small proof-of-concept virtual memory manager using the Coq Proof Assistant.

The rest of this paper is organized as follows. In Section 2, we give an informal overview of our work. In Section 3, we discuss the formal details of our verification and refinement framework. In Section 4, we specialize the framework for a simple C-like language. In Section 5, we certify BabyVMM, our small virtual memory manager. Section 6 discusses the Coq proof, and Section 7 presents related work and concludes.

2 Overview and Plan for Certification

We begin the overview by explaining the design of BabyVMM, our small virtual memory manager. First, consider the model of memory present in simplified hardware (left side of Figure 1). The memory is a storage system, which contains cells that can be read from or written to by the software. These cells are indexed by addresses. However, to facilitate indirection, the hardware includes a system called address translation (AT), which, when enabled, will cause all requests for specific addresses from the software to be translated. The AT system adds special registers to the memory system - one to enable or disable AT, and the other to point where the software-managed AT tables are located in memory. The fact that these tables are stored in memory is one of the sources of complexity in the AT system - updating AT tables requires updating in-memory tables, a process which goes through AT as well.

Fig. 1. Hardware (HW) and Address Space (AS) Models of Memory

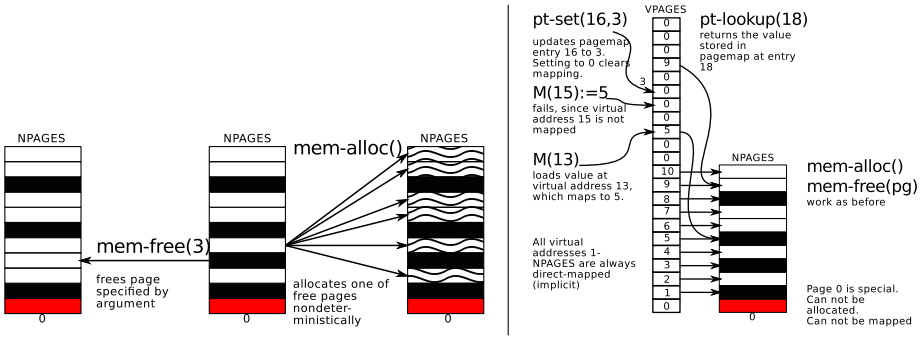


Fig. 2. Allocated (ALE) and Page Map (PMAP) Models of Memory

Because AT is such a complicated, machine-dependent, and general mechanism, BabyVMM creates an abstraction that defines specific restrictions on how AT will be used, and presents a simpler view of AT to the kernel. Although the abstract models of memory may differ depending on the features that the kernel may require, BabyVMM defines a very basic model, to which we refer as the address space (AS) model of memory (right side of Figure 1). The AS model replaces the small physical memory with a larger virtual address space with allocatable pages and no address translation. The space is divided into high and low areas, where the low area is actually a window into physical memory (a pattern common in many kernels). Because of this distinction, the memory model has two sets of allocation functions, one for the "high" memory area where the programmer requests a specific page for allocation, and one for the "low" memory area, where the programmer can not pick which page to allocate.

However, creating an abstraction that makes the jump from the HW model directly to AS model is complex. As a result, we create two more intermediate models, which slowly build up the abstraction. The first model is ALE (left side of Figure 2), which incorporates allocation information into the hardware memory, requiring that programs only access memory locations that are marked allocated. The model adds primitives in the form of `mem_alloc` and `mem_free`, with semantics same as the ones in the AS

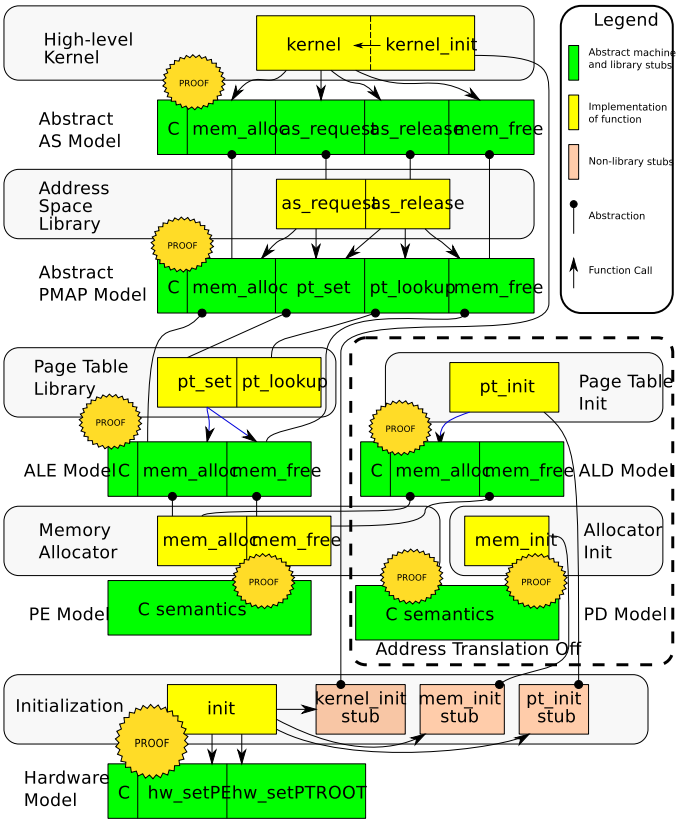


Fig. 3. Complete Plan for VMM Certification

model. Although this is not shown on the diagram, the ALE model still maintains the hardware's AT mechanism.

The second intermediate level, which we call PMAP (right side of Figure 2) is designed to replace the hardware's AT mechanism with an abstract one. The model features a page map that exists outside the normal memory space, unlike the lower level models. The page map maps virtual page numbers to physical page numbers, with a 0 value meaning invalid. In our particular design, the pagemap is always identity for the lower addresses, creating a window into physical memory from within the virtual space. The model still contains allocation primitives, and adds two more primitives, `pt_set` and `pt_lookup`, which update and lookup values in the pagemap.

Using these abstract memory models, we can construct the BabyVMM verification plan (Figure 3). The light-yellow boxes in the kernel represent the actual functions (actual code is given in Appendix A of TR[19]). The darker green boxes represent computational models with primitives labeled. The diagram shows how each module of BabyVMM will be certified in the model best suited for it. For example, the high-level kernel is certified in the AS model, meaning that it does not see underlying physical memory at all. The implementation of `as_request` and `as_release` are defined over

$$
\begin{array}{rlrl}
(\textit{State})\ \mathbb{S} & \in \Sigma & (\textit{State Predicate})\ \mathtt{p} & \in \Sigma \to \textit{Prop} \\
(\textit{Operation})\ \iota & \in \Delta & (\textit{State Relation})\ \mathtt{g} & \in \Sigma \to \Sigma \to \textit{Prop} \\
(\textit{Cond})\ b & \in \beta & (\textit{Operational Semantics})\ \mathtt{OS} & \in \{\iota \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* \\
(\textit{CondInterp})\ \Upsilon & \in \beta \to \Sigma \to \textit{Prop} & (\textit{Language / Machine})\ \mathcal{M} & \in (\Sigma,\Delta,\beta,\Upsilon,\mathtt{OS})
\end{array}
$$

where $\mathcal{M}(\iota) \triangleq \mathcal{M}.\mathtt{OS}(\iota)$ and $\mathcal{M}(b) \triangleq \mathcal{M}.\Upsilon(b)$

Fig. 4. Abstract State Machine

$$
\begin{array}{ll}
\textit{id} & \triangleq (\lambda\mathbb{S}.\,\mathsf{True}, \quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathbb{S}' = \mathbb{S}) \\
\textit{fail} & \triangleq (\lambda\mathbb{S}.\,\mathsf{False}, \quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathsf{False}) \\
\textit{loop} & \triangleq (\lambda\mathbb{S}.\,\mathsf{True}, \quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathsf{False}) \\
(\mathtt{p},\mathtt{g}) \circ (\mathtt{p}',\mathtt{g}') & \triangleq (\lambda\mathbb{S}.\,\mathtt{p}\ \mathbb{S} \wedge \forall\mathbb{S}'.\,\mathtt{g}\ \mathbb{S}\ \mathbb{S}' \to \mathtt{p}'\ \mathbb{S}', \quad \lambda\mathbb{S}.\lambda\mathbb{S}''.\,\exists\mathbb{S}'.\,\mathtt{g}\ \mathbb{S}\ \mathbb{S}' \wedge \mathtt{g}'\ \mathbb{S}'\ \mathbb{S}'') \\
(\mathtt{p},\mathtt{g}) \underset{c}{\oplus} (\mathtt{p}',\mathtt{g}') & \triangleq (\lambda\mathbb{S}.\,(\mathtt{p}\ \mathbb{S} \wedge c\ \mathbb{S}) \vee (\mathtt{p}'\ \mathbb{S} \wedge \neg c\ \mathbb{S}), \lambda\mathbb{S}.\lambda\mathbb{S}'.\,(c\ \mathbb{S} \wedge \mathtt{g}\ \mathbb{S}\ \mathbb{S}') \vee (\neg c\ \mathbb{S} \wedge \mathtt{g}'\ \mathbb{S}\ \mathbb{S}')) \\
(\mathtt{p},\mathtt{g}) \supseteq (\mathtt{p}',\mathtt{g}') & \triangleq \forall\mathbb{S}.\,\mathtt{p}\ \mathbb{S} \to \mathtt{p}'\ \mathbb{S} \wedge \forall\mathbb{S},\mathbb{S}'.\,\mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \to \mathtt{g}\ \mathbb{S}\ \mathbb{S}'
\end{array}
$$

Fig. 5. Combinators and Properties of Actions

$$
\begin{array}{rlll}
(\textit{Meta-program}) & \mathbb{P} & ::= (\mathbb{C},\mathbb{I}) & [\![\mathbb{C},\mathtt{a}]\!]^0_{\mathcal{M}} := \textit{loop} \\
(\textit{Proc}) & \mathbb{I} & ::= \mathtt{nil} \mid \iota \mid [\mathtt{l}] \mid \mathbb{I}_1;\mathbb{I}_2 & [\![\mathbb{C},\mathtt{nil}]\!]^n_{\mathcal{M}} := \textit{id} \\
 & & \mid (b?\ \mathbb{I}_1 + \mathbb{I}_2) & [\![\mathbb{C},\iota]\!]^n_{\mathcal{M}} := (\mathcal{M}(\iota)) \\
(\textit{Proc Heap}) & \mathbb{C} & ::= \{\mathtt{l} \rightsquigarrow \mathbb{I}\}^* & [\![\mathbb{C},[\mathtt{l}]]\!]^n_{\mathcal{M}} := [\![\mathbb{C},\mathbb{C}(\mathtt{l})]\!]^{n-1}_{\mathcal{M}} \\
(\textit{Labels}) & \mathtt{l} & ::= n \text{ (nat numbers)} & [\![\mathbb{C},\mathbb{I};\mathbb{I}']\!]^n_{\mathcal{M}} := [\![\mathbb{C},\mathbb{I}]\!]^n_{\mathcal{M}} \circ [\![\mathbb{C},\mathbb{I}]\!]^n_{\mathcal{M}} \\
(\textit{Spec Heap}) & \Psi,\mathcal{L} & ::= \{\mathtt{l} \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* & [\![\mathbb{C},(b?\ \mathbb{I}_1+\mathbb{I}_2)]\!]^n_{\mathcal{M}} := [\![\mathbb{C},\mathbb{I}_1]\!]^n_{\mathcal{M}} \underset{\mathcal{M}(b)}{\oplus} [\![\mathbb{C},\mathbb{I}_2]\!]^n_{\mathcal{M}}
\end{array}
$$

Fig. 6. Syntax and Semantics of the Meta-Language

an abstract page map, and thus do not have to know how the hardware deals with page tables, and so on. The plan also indicates which primitives are implemented by which code (lines with circles). When we certify the code, these will be the cross-abstraction links we will have to prove. Lastly, the plan also indicates the stubs in the initialization, which are needed to certify calls from init to functions defined over higher abstraction. The PE and PD models are restrictions on HW model, where AT is always on, and always off respectively. ALD is an analogue of ALE, where AT is off.

On boot, the AT is off, and `init` is called. The `init` then calls `mem_init` to initialize the allocation table and `pt_init` to initialize the page tables. Then, `init` uses the HW primitives to enable AT, and jumps into the high-level kernel by calling `kernel_init`.

We will now focus on the technical details to put this plan in action.

3 Certifying with Refinement

Our framework for multi-machine certification is defined in two parts. First, we create a machine-independent verification framework that will allow us to define quickly and easily as many machines for verification as we need. Second, we will develop our notion of refinements which will allow us to link all the separate machines together.

$$\frac{\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}).\, \mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})}{\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi}\ (\textsc{code}) \qquad \frac{\mathcal{M}, \Psi \vdash \mathbb{I} : (\mathtt{p}', \mathtt{g}') \quad (\mathtt{p}, \mathtt{g}) \supseteq (\mathtt{p}', \mathtt{g}')}{\mathcal{M}, \Psi \vdash \mathbb{I} : (\mathtt{p}, \mathtt{g})}\ (\textsc{weak})$$

$$\frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (\mathtt{p}', \mathtt{g}') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (\mathtt{p}'', \mathtt{g}'')}{\mathcal{M}, \Psi \vdash (b?\, \mathbb{I}' + \mathbb{I}'') : (\mathtt{p}', \mathtt{g}') \underset{\mathcal{M}(b)}{\oplus} (\mathtt{p}'', \mathtt{g}'')}\ (\textsc{split}) \quad \frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (\mathtt{p}', \mathtt{g}') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (\mathtt{p}'', \mathtt{g}'')}{\mathcal{M}, \Psi \vdash \mathbb{I}'; \mathbb{I}'' : ((\mathtt{p}', \mathtt{g}') \circ (\mathtt{p}'', \mathtt{g}''))}\ (\textsc{seq})$$

$$\frac{}{\mathcal{M}, \Psi \vdash \iota : \mathcal{M}(\iota)}\ (\textsc{perf}) \qquad \frac{}{\mathcal{M}, \Psi \vdash [\mathtt{l}] : \Psi(\mathtt{l})}\ (\textsc{call}) \qquad \frac{}{\mathcal{M}, \Psi \vdash \mathtt{nil} : id}\ (\textsc{nil})$$

Fig. 7. Static Semantics of the Meta-Language

3.1 A Machine-Independent Certification Framework

Our Hoare-logic based framework is parametric over the definition of operational semantics of the machine, and is sound no matter what machine semantics it is parameterized with. To begin defining such a framework, we first need to understand what exactly is a machine on which we can certify code. The definition that we use is given in Figure 4. Our notion of the machine consists of the following parts:

- State type (Σ). Define the set of all possible states in a machine.
- Operations (Δ). This is a set of names of all operations that the machine supports. The set can be infinite, and defined parametrically.
- Conditionals (β). Defines a type of expressions that are used for branching.
- Conditional Interpreter (Υ). Converts conditionals into state predicates.
- The operational semantics $\mathtt{OS}$. This is the main portion of the machine definition. It is a set of actions $(\mathtt{p}, \mathtt{g})$ named by all operations in the machine.

The most important bit of information in the machine are the semantics ($\mathtt{OS}$). The semantics of operations are defined by a precondition ($\mathtt{p}$), which shows when the operation is safe to execute, and by a state relation ($\mathtt{g}$) that defines the set of possible states that the operation may result in. We will refer to the pair of $(\mathtt{p}, \mathtt{g})$ as an action of the operation. Later we will also use actions to define the specification of programs. Because the type of actions is somewhat complex, we define action combinators in Figure 5, including composition and branching. The same figure also shows the weaker than relation between actions.

Although, at this point we have defined our machines, it does not have any notion of computation. To make use of the machine, we will need to define a concept of programs, as well as what it means for the particular program to execute.

The definition of the program is given in Figure 6. The most important definition in that figure is that of the procedure, $\mathbb{I}$. The procedure is a bit of program logic that sequences together calls to the operations of a machine (ι), or to other procedures $[\mathtt{l}]$ (loops are implemented as recursive calls). Procedures also include a way to branch on a condition. The procedures can be given a name, and placed in the procedure heap $\mathbb{C}$, where they can be referenced from other procedures through the $[\mathtt{l}]$ constructor. The procedure heap together with a program rest (the currently executing procedure) makes up the program that can be executed.

The meaning of executing a program is given by the indexed denotational semantics shown on the right side of Figure 6. The meaning of the program is an action that is

constructed by sequencing operations. As programs can be infinite, the semantics are indexed by the depth of procedure inclusion.

We use the static semantics (Figure 7) to approximate the action of a procedure. These semantics are similar to the denotational semantics of the meta-language, except that the specifications of called procedure are looked up in the table (Ψ). This means that the static semantics works by the programmer approximating the actions of (specifying) the program, and then making sure that the actual action of the program is within the specifications. These well-formed procedures are then grouped into a well-formed module using the CODE rule, which forms the concept of a certified module $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, where every procedure in $\mathbb{C}$ is well-formed under specification in Ψ. The module also defines a library ($\mathcal{L}$) which is a set of specifications of stubs, i.e. procedures that are used by the module, but are not in the module. These stubs can then be eliminated by providing procedures that match the stubs (see Section 3.2). For a program to be completely certified, all stubs must either be considered valid primitives or eliminated.

For a proof of partial correctness, please see the TR.

3.2 Linking

When we certify using modules, it will be very common that the module will require stubs for the procedures of another module. Linking two modules together should replace the stubs in both modules for the actual procedures that are now present in the linked code. The general way to accomplish this is by the following linking lemma:

Theorem 1 (Linking).

$$\frac{\mathcal{M}, \mathcal{L}_1 \vdash \mathbb{C}_1 : \Psi_1 \quad \mathcal{M}, \mathcal{L}_2 \vdash \mathbb{C}_2 : \Psi_2 \quad \mathbb{C}_1 \perp \mathbb{C}_2 \quad \mathcal{L}_1 \perp \Psi_2 \quad \mathcal{L}_2 \perp \Psi_1 \quad \mathcal{L}_1 \perp \mathcal{L}_2}{\mathcal{M}, ((\mathcal{L}_1 \cup \mathcal{L}_2) \setminus (\Psi_1 \cup \Psi_2)) \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2} \text{ (LINK)}$$

where $\Psi_1 \perp \Psi_2 \triangleq \forall \mathtt{l} \in \mathtt{dom}(\Psi_1).\ (\mathtt{l} \notin \mathtt{dom}(\Psi_2) \vee \Psi_1(\mathtt{l}) = \Psi_2(\mathtt{l}))$.

However, the above rule does not always apply immediately. When the two modules are developed independently, it is possible that the stubs of one module are weaker than the specifications of the procedures that will replace the stubs, which breaks the linking lemma. To fix this, we strengthen the library.

Theorem 2 (Stub Strengthening).
If $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, then for any $\mathcal{L}'$ s.t. $\forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}).\ \mathcal{L}(\mathtt{l}) \supseteq \mathcal{L}'(\mathtt{l})$ and $\mathtt{dom}(\mathcal{L}') \cap \mathtt{dom}(\Psi) = \emptyset$, the following holds: $\mathcal{M}, \mathcal{L}' \vdash \mathbb{C} : \Psi$.

This theorem allows us to strengthen the stubs to match the specs of procedures, enabling the linking lemma. Of course, if the specs of the real procedures are not stronger than the specs of the stubs, then the procedures do not correctly implement what the module expects, and linking is not possible.

3.3 The Refinement Framework

Up to this point, we have only considered what happens to the code that is certified over a single machine. However, the purpose of our framework is to facilitate multi-machine

verification. For this purpose, we construct the refinement framework that will allow us to refine certified modules in one machine to certified modules in another. The most general notion of refinement in our framework can be defined by the following:

Definition 1 (Certified Refinement).
A certified refinement from machine $\mathcal{M}_A$ to machine $\mathcal{M}_C$ is a pair of relations $(T_{\mathbb{C}}, T_\Psi)$ and a predicate over the abstract certified module Acc, such that for all $\mathbb{C}_A, {\Psi'}_A, \Psi_A$, the following holds

$$\frac{\mathcal{M}_A, {\Psi'}_A \vdash \mathbb{C}_A : \Psi_A \quad Acc(\mathcal{M}_A, {\Psi'}_A \vdash \mathbb{C}_A : \Psi_A)}{\mathcal{M}_C, T_\Psi({\Psi'}_A) \vdash T_{\mathbb{C}}(\mathbb{C}_A) : T_\Psi(\Psi_A)}\ \textsc{refine}$$

This definition is not a rule, but a template for other definitions. To define a refinement, one has to provide the particular $T_{\mathbb{C}}$, T_Ψ, Acc together with the proof that the rule holds. However, instead of trying to define these translations directly, we will automatically generate them from the relations between the particular pairs of machines.

Representation Refinement The only automatic refinement we will discuss in this paper is the representation refinement. The representation refinement can be generated for an abstract ($\mathcal{M}_A$) and a concrete machine ($\mathcal{M}_C$), where both use the same operations and condtionals (e.g. $\mathcal{M}_A.\Delta = \mathcal{M}_C.\Delta$ and $\mathcal{M}_A.\beta = \mathcal{M}_C.\beta$) by defining a relation ($\texttt{repr} : \mathcal{M}_A.\Sigma \to \mathcal{M}_C.\Sigma \to \mathit{Prop}$) between the states of the two machines. Using `repr`, we can define our specification translation function:

$$T_{A-C}(\mathtt{p}, \mathtt{g}) \triangleq \begin{array}{l}(\lambda \mathbb{S}_C.\, \exists \mathbb{S}_A.\, \mathtt{repr}\ \mathbb{S}_A\ \mathbb{S}_C \wedge \mathtt{p}\ \mathbb{S}_A, \\ \quad \lambda \mathbb{S}_C.\, \lambda \mathbb{S}'_C.\, \forall \mathbb{S}_A.\, \mathtt{repr}\ \mathbb{S}_A\ \mathbb{S}_C \to \forall \mathbb{S}'_A.\, \mathtt{g}\ \mathbb{S}_A\ \mathbb{S}'_A \to \mathtt{repr}\ \mathbb{S}'_A\ \mathbb{S}'_C)\end{array}$$

This operation creates an concrete action from an abstract action. Informally it works as follows. There must be at least one abstract state related to the starting concrete state for which the abstract action applies. The action starting from state $\mathbb{S}_C$ results in set containing $\mathbb{S}'_C$, only if for all related abstract states for which the abstract action is valid result in sets of abstract states that contain a state related to $\mathbb{S}'_C$. Essentially, the resulting concrete action is an intersection of all abstract actions that do not fail.

To make this approach work, we require several properties over the machines and the `repr`. First, the refined semantics of abstraction operations have to be weaker than the semantics of their concrete counterparts, e.g. $\forall \iota_A \in \mathcal{M}_A.\, T_{A-C}(\mathcal{M}_A(\iota_A)) \supseteq \mathcal{M}_C(\iota_A)$.

Second, the refinement must preserve the branch choice, e.g. if the refined program chooses left branch, then abstract program had to choose the left branch in all states related by `repr` as well. This property is ensured by requiring the following:

$$\forall b.\, \forall \mathbb{S}, \mathbb{S}'.\, (\exists \mathbb{S}_C.\, \mathtt{repr}(\mathbb{S}, \mathbb{S}_C) \wedge \mathtt{repr}(\mathbb{S}', \mathbb{S}_C)) \to (\mathcal{M}(b)\ \mathbb{S} \leftrightarrow \mathcal{M}(b)\ \mathbb{S}')$$

With these properties, we can define a valid refinement by the following lemma:

Lemma 1 (`repr`-refinement valid).
Given `repr` with proofs of the two properties above, the following is valid:

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi_A}{\mathcal{M}_C, T_\Psi(\mathcal{L}_A) \vdash \mathbb{C} : T_\Psi(\Psi_A)}$$

where $T_\Psi(\Psi) := \{T_{A-C}(\Psi(\mathtt{l})) \mid \mathtt{l} \in \mathtt{dom}(\Psi)\}$

This refinement is interesting in that it preserves the code of the program, and performing point-wise refinement on specifications. Our actual work defines several other refinement generators. One of these, code-preserving refinement, is included in the TR, and is used as a stepping stone for proof of Lemma 1. Coq implementation features more general versions of refinements presented, as well as several others.

4 Certifying C Code

Since BabyVMM is written in C, we define a formal specification of a tiny subset of the C language using our framework. This C machine will be parameterized by the specific semantics of the memory model, as our plan required. We will also utilize the C machine to further speed up the creation of refinements.

4.1 The Semantics of C

To define our C machine in terms of our verification framework, we need to give it a state type, a list of operations, and the semantics of those operations expressed as actions. All of these are given in Figure 8.

The state of the C machine includes two components, the stack and the memory. The stack is an abstract C stack that consists of a list of frames, which include call, data, and return frames. In the current version, the stack is independent from memory (one can think of it existing within a statically defined part of the loaded kernel). The memory model is a parameter in the C machine, meaning that it can make use of any memory model as long as it defines load and store operations. The syntax of the C machine is different from the usual definition, in that it relies on the meta-machine for its control flow by using the meta-machines call and branch. Our definition of C adds atomic operations that perform state updates. Thus the operations include two types assignments - one to stack and one to memory, and 4 operations to manipulate stack for call and return, which push and pop the frames.

Because control flow is provided by a standard machine, the code has to be modified slightly. For example, a function call of the form $r = f(x)$ will split into a sequence of three operations: $fcall([x]); [f]; readret([r])$, the first setting up a call frame, the second making the call, and the third doing the cleanup. Similarly, the body of the function $f(x)\{body; return(0);\}$ will become $args([x]); body; ret(0)$, as the function must first move the arguments from the call frame into a data frame. Loops have to be desugared into recursive procedures with branches. These modifications are entirely mechanical, and hence we can claim that our machine supports desugared linearized C code.

4.2 Refinement in C machines

C machines at different abstraction layers differ only in their memory models, with the stack being the same. We can use this fact to generate refinements between the C machines using only the representation relation between memory models. This relation ($M_1 \preceq M_2$) can be completely arbitrary as long as these conditions hold:

$$\forall l, v.\, load(M_1, l) = v \rightarrow load(M_2, l) = v$$
$$\forall l, v, M_1'.\, \left(M_1' = (store(M_1, l, v))\right) \rightarrow \left(M_1' \preceq (store(M_2, l, v))\right)$$

$$
\begin{array}{rll}
(\textit{State}) & \mathbb{S} & ::= (M, S) \\
(\textit{Memory}) & M & ::= \textit{(any type over which load}(M,\mathtt{l})\textit{ and store}(M,\mathtt{l},w)\textit{ are defined)} \\
(\textit{Stack}) & S & ::= \textit{nil} \mid \textit{Call}(\textit{list } w) :: S \mid \textit{Data}(\{\mathtt{v} \rightsquigarrow w\} :: S) \mid \textit{Ret}(w) :: S \\
(\textit{Expressions}) & \mathtt{e} & ::= \mathrm{se} \mid *(\mathtt{e}) \\
(\textit{StackExpr, Cond}) & \mathrm{se}, b & ::= w \mid \mathtt{v} \mid \textit{binop}(\textit{bop}, \mathtt{e}_1, \mathtt{e}_2) \\
(\textit{Binary Operators}) & \textit{bop} & ::= + \mid - \mid * \mid / \mid \% \mid == \mid < \mid <= \mid >= \mid > \mid != \mid \&\& \mid \mid\mid \\
(\textit{Variables}) & \mathtt{v} & ::= \textit{(a decidable set of names)} \\
(\textit{Words}) & w & ::= n \text{ (integers)} \\
(\textit{Operation}) & \iota & ::= \mathtt{v} := \mathtt{e} \mid *(\mathtt{e}_{loc}) := \mathtt{e} \mid \mathrm{fcall}(\mathrm{list}\ \mathtt{e}) \mid \mathrm{ret}(\mathtt{e}) \mid \mathrm{args}(\mathrm{list}\ \mathtt{v}) \mid \mathrm{readret}(\mathtt{v})
\end{array}
$$

Operation (ι) =	Action ($\mathcal{M}(\iota)$) =
$\mathtt{v} := \mathtt{e}$	$(\lambda\mathbb{S}.\, \exists S', F, w.\, \mathbb{S}.S = \textit{Data}(F) :: S' \wedge \textit{eval}(\mathtt{e}, \mathbb{S}) = w,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', F, w.\, \mathbb{S}.S = \textit{Data}(F) :: S' \wedge \textit{eval}(\mathtt{e}, \mathbb{S}) = w \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Data}(F\{v \rightsquigarrow w\}) :: S')$
$*(\mathtt{e}_{loc}) := \mathtt{e}$	$(\lambda\mathbb{S}.\, \exists l, w.\, \textit{eval}(\mathtt{e}, \mathbb{S}) = w \wedge \textit{eval}(\mathtt{e}_{loc}, \mathbb{S}) = l \wedge \exists M'.\, M' = \textit{store}(M, l, w),$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists l, w.\, \textit{eval}(\mathtt{e}, \mathbb{S}) = w \wedge \textit{eval}(\mathtt{e}_{loc}, \mathbb{S}) = l \wedge$ $\quad \mathbb{S}'.M = \textit{store}(\mathbb{S}.M, l, w) \wedge \mathbb{S}'.S = \mathbb{S}.S)$
$\mathrm{fcall}([e_1, \ldots, e_n])$	$(\lambda\mathbb{S}.\, \exists v_1, \ldots, v_n.\, \textit{eval}(e_1, \mathbb{S}) = v_1 \wedge \ldots \wedge \textit{eval}(e_n, \mathbb{S}) = v_n,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists v_1, \ldots, v_n.\, \textit{eval}(e_1, \mathbb{S}) = v_1 \wedge \ldots \wedge \textit{eval}(e_n, \mathbb{S}) = v_n \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Call}([v_1, \ldots, v_n]) :: \mathbb{S}.S)$
$\mathrm{args}([\mathtt{v}_1, \ldots, \mathtt{v}_n])$	$(\lambda\mathbb{S}.\, \exists w_1, \ldots, w_n, S'.\, \mathbb{S}.S = \textit{Call}([w_1, \ldots, w_n]) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists w_1, \ldots, w_n, S'.\, \mathbb{S}.S = \textit{Call}([w_1, \ldots, w_n]) :: S' \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Data}(\{\mathtt{v}_1 \rightsquigarrow w_1, \ldots, \mathtt{v}_n \rightsquigarrow w_n\}) :: S')$
$\mathrm{readret}(\mathtt{v})$	$(\lambda\mathbb{S}.\, \exists S', w.\, \mathbb{S}.S = \textit{Ret}(w) :: \textit{Data}(D) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', w.\, \mathbb{S}.S = \textit{Ret}(w) :: \textit{Data}(D) :: S' \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Data}(D\{\mathtt{v} \rightsquigarrow w\}) :: S')$
$\mathrm{ret}(\mathtt{e})$	$(\lambda\mathbb{S}.\, \exists w.\, \textit{eval}(\mathtt{e}, \mathbb{S}) = w,\ \lambda\mathbb{S}, \mathbb{S}'.\, \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Ret}(\textit{eval}(\mathtt{e}, \mathbb{S})) :: \mathbb{S}.S)$

$$
\textit{eval}(\mathtt{e}, \mathbb{S}) ::= \begin{cases} w & \text{if } \mathtt{e} = w \\ \mathbb{S}.S(\mathtt{v}) & \text{if } \mathtt{e} = \mathtt{v} \\ \textit{load}(\mathbb{S}.M, \textit{eval}(\mathtt{e}_1, \mathbb{S})) & \text{if } \mathtt{e} = (*\mathtt{e}_1) \\ b(\textit{eval}(\mathtt{e}_1, \mathbb{S}), \textit{eval}(\mathtt{e}_2, \mathbb{S})) & \text{if } \mathtt{e} = \textit{binop}(b, \mathtt{e}_1, \mathtt{e}_2) \end{cases}
$$

$$
\Upsilon(b) ::= \lambda\mathbb{S}.\, \textit{eval}(b, \mathbb{S}) \neq 0
$$

Fig. 8. Primitive C-like machine

The above properties make sure that the load and store operations of memory behave in a similar way. We construct the `repr` between C machine as follows:

$$
\mathtt{repr} := \lambda\mathbb{S}_A, \mathbb{S}_C.\ (\mathbb{S}_A.S = \mathbb{S}_C.S)\ \wedge\ (\mathbb{S}_A.M \preceq \mathbb{S}_C.M)
$$

Using the properties of load and store, we show properties needed for `repr`-refinement to work: that for every operation ι in the C machine $T_{M1-M2}(\mathcal{M}_{M1}(\iota)) \supseteq \mathcal{M}_{M2}(\iota)$, and that `repr` preserves branching. For details, please see the TR. Now we can define the actual refinement rule for C machines:

Corollary 1 (C Refinement).
For any two memory models $M1$ and $M2$, s.t. $M1 \preceq M2$, the following refinement works for C

Definition	Value	Description
PGSIZE	4096	Number of bytes per page
NPAGES	unspecified	Number of phys. pages in memory
VPAGES	unspecified	Maximum page number of a virtual address
Pg(*addr*)	*addr*/PGSIZE	gets page of address
Off(*addr*)	*addr*%PGSIZE	offset into page of address
LowPg(*pg*)	$0 \leq pg <$ NPAGES	valid page in low memory area
HighPg(*pg*)	NPAGES $\leq pg <$ VPAGES	valid page in high memory area

Fig. 9. Page Definitions

machines instantiated with $M1$ and $M2$.

$$\frac{\mathcal{M}_{M1}, \mathcal{L} \vdash \mathbb{C} : \Psi}{\mathcal{M}_{M2}, T_{M1-M2}(\mathcal{L}) \vdash \mathbb{C} : T_{M1-M2}(\Psi)} \; M1 - M2$$

Thus we know that if we have two C-machines that have related memory models, then we have a working refinement between the two machines. Our next step is the to show the relations between all the memory models shown in our plan (in Figure 3).

5 Virtual Memory Manager

At this point, we have all the machinery necessary to start building our certified memory manager according to the plan. The first step is to formally define and give relations between the memory models that we will use in our certification. Then we will certify the code of the modules that make up the VMM. These modules will then be refined and linked together, resulting in the conclusion that the entire BabyVMM is certified.

5.1 The Memory Models

Because of the space limit, we will only formally present the PMAP memory model (Figures 9 and 10). For the definitions of others, please see the TR.

The state of the PMAP memory has three components, the actual memory store D, the allocation table A, and the first-class pagemap PM. The memory store contains the actual data in memory, indexed by physical addresses. The allocation table A, keeps track of which pages are allocated and which are not. This allocation information is abstract - it does not have to correspond to the actual allocation table used within the VMM. For example, the hardware page tables, which this model abstracts, are still in memory, but are hidden by the allocation table. The page map is the abstract mapping of virtual pages to physical pages, which purposefully skips all addresses mappable to physical memory. This mapping is used in loads and stores of the memory model, which use the *trans* predicate to translate addresses by looking up mappings in the PM.

The PMAP model relies on the stub library ($\mathcal{L}_{PMAP}$) for updating auxiliary data structures. There are two stubs for memory allocation, mem_alloc and mem_free. Their specs show how they modify the allocation table, and how allocating a page is non-deterministic and may potentially return any free page. The other two stubs, pt_set and pt_lookup update and look up page map entries; their specs are straightforward.

$$
\begin{array}{rll}
\text{(Global Storage System)} & M & ::= (D, A, PM) \\
\text{(Allocatable Memory)} & D & ::= \{addr \rightsquigarrow w \mid \text{LowPg}(Pg(addr)) \wedge addr\%8 = 0\}^* \\
\text{(Page Allocation Table)} & A & ::= \{pg \rightsquigarrow \texttt{bool} \mid \text{LowPg}(pg)\}^* \\
\text{(Page Map)} & PM & ::= \{pg \rightsquigarrow pg' \mid \text{HighPg}(pg)\}^*
\end{array}
$$

Notation	Definition	
$load(M, va)$	$M.D(trans(M, va))$	if $M.A(Pg(trans(M, va))) = true$
$store(M, va, w)$	$(M.D\{trans(M, va) \rightsquigarrow w\}, M.A, M.PM)$	if $M.A(Pg(trans(M, va))) = true$

$$
trans(M, va) := \begin{cases} M.PM(Pg(va)) * \texttt{PGSIZE} + \text{Off}(va) & \text{if HighPg}(Pg(va)) \\ va & \text{otherwise} \end{cases}
$$

Label	Specification
mem_alloc	$(\lambda\mathbb{S}.\, \exists S'.\, \mathbb{S}.S = Call([]) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S'.\, (\mathbb{S}.S = Call([]) :: S') \wedge ((\mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M) \vee$ $(\exists pg.\, \mathbb{S}'.S = Ret(pg) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \rightsquigarrow true\} \wedge \mathbb{S}'.M.PM = \mathbb{S}.M.PM \wedge$ $\wedge \mathbb{S}.M.A(pg) = false \wedge \forall l.\, \mathbb{S}.M.A(Pg(l)) = true \rightarrow (\mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))))$
mem_free	$(\lambda\mathbb{S}.\, \exists S', pg.\, \mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}.M.A(pg) = true,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', pg.\, \mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M.PM = \mathbb{S}.M.PM \wedge$ $\mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \rightsquigarrow false\} \wedge \forall l.\, \mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
pt_set	$(\lambda\mathbb{S}..\exists S', vp, pp.\, \mathbb{S}.S = Call([vp, pp]) :: S' \wedge \text{HighPg}(vp) \wedge \text{LowPg}(pp)$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', vp, pp.\, \mathbb{S}.S = Call([vp, pp]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A \wedge$ $\mathbb{S}'.M.PM = \mathbb{S}.M.PM\{vp \rightsquigarrow pp\} \wedge \forall l.\, \mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
pt_lookup	$(\lambda\mathbb{S}.\, \exists S', vp.\, \mathbb{S}.S = Call([vp]) :: S' \wedge \text{HighPg}(vp),$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', vp.\, \mathbb{S}.S = Call([vp]) :: S' \wedge \mathbb{S}'.S = Ret(\mathbb{S}.M.PM(vp)) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M)$

Fig. 10. PMAP Memory Model (M_{PMAP}) and Library ($\mathcal{L}_{PMAP}$)

5.2 Relation between Memory Models

Our plan calls for creation of the refinements between the memory models. In Section 4.2, we have shown that we can generate a valid refinement by creating a relation between the memory states, and then showing that abstract loads and stores are preserved by this relation. These relations and proofs of preserving the memory operations are fairly lengthy and quite technical, and thus we leave the mathematical detail to our Coq implementation, opting for a visual description shown in Figure 11.

On the right is a state of the hardware memory, whose operational semantics gives little protection from accessing data. Some areas of memory are dangerous, some are empty, others contain data, including the allocation tables and page tables. This memory relates to the ALE memory model by abstracting out the memory allocation table. This allocation table now offers protection for accessing both the unallocated space, and the space that seems unallocated, but dangerous to use (marked by wavy lines). An example of such area is the allocation table itself - the ALE model hides the table, making it appear to be unusable. The ALE mem_alloc primitive will never allocate pages from these wavy areas, protecting them without complicating the memory model.

The relation between the PMAP and ALE models shows that the abstract pagemap of PMAP model is actually contained within the specific area of the ALE model. The relation makes sure that the mappings contained in the PMAP's pagemap are the same as the translation results of the ALE's page table structures. To protect the in memory

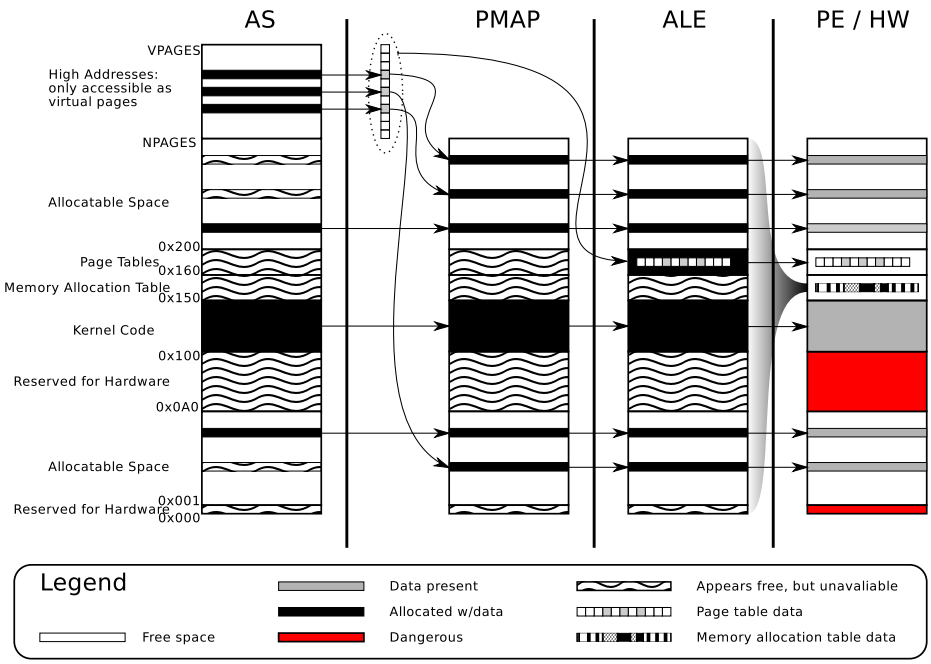


Fig. 11. Relation between Memory Models

page tables, the relation hides the page table memory area from the PMAP model, using the same trick as the one used to protect the allocation tables in the ALE model.

The relation between the AS and PMAP models collapses PMAP's memory and the page maps into a single memory like structure in the AS model. This is mostly accomplished by chaining the translation mechanism with the storage mechanism. However, to make this work, it is imperative that the relation ensures that no two pages of the AS model ever map to the same physical page in the PMAP model. This means that all physical pages that are mapped from the high-addresses become hidden in the AS model. We will not go into detail about the preservation of load and stores, as these proofs are mostly straightforward, given the relations.

5.3 Certification and Linking of BabyVMM

We have verified all the functions of the virtual memory on the appropriate memory models. This means that we have defined appropriate specifications for our functions, and certified our code. We also make an assumption that a kernel is certified in the AS model. The result is the following certified modules:

$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem} : \Psi^{mem}_{PE} \quad \mathcal{M}_{ALE}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi^{as}_{PMAP} \quad \mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{meminit} : \Psi^{meminit}_{PD}$$
$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi^{pt}_{ALE} \quad \mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi^{kernel}_{AS} \quad \mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{ptinit} : \Psi^{ptinit}_{ALD}$$

However, the `init` function makes calls to other procedures that are certified in more abstract machines. Thus to certify `init` over the $\mathcal{M}_{HW}$ machine, we will need to

create stubs for these procedures, which have to be carefully crafted to be valid for the refined specifications of the actual procedures. Thus, the specification of `init` results in the following:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \left\{\text{kernel\_init} \rightsquigarrow \mathtt{a}_{HW}^{kernel-init}, \text{mem\_init} \rightsquigarrow \mathtt{a}_{HW}^{meminit}, \text{pt\_init} \rightsquigarrow \mathtt{a}_{HW}^{ptinit}\right\} \vdash \mathbb{C}^{init} : \Psi_{HW}^{init}$$

With all the modules verified, we proceed to link them together. The first step is to refine the kernel. We use our AS-PMAP refinement rule to get the refined module:

$$\mathcal{M}_{PMAP}, T_{AS-PMAP}(\mathcal{L}_{AS}) \vdash \mathbb{C}^{kernel} : T_{AS-PMAP}(\Psi_{AS}^{kernel})$$

Then we show that the specs of functions and the primitives of the PMAP machine are proper implementation of the refined specs of $\mathcal{L}_{AS}$, more formally, $T_{AS-PMAP}(\mathcal{L}_{AS}) \supseteq \mathcal{L}_{PMAP} \cup \Psi_{PMAP}^{as}$. Using library strengthening and the linking lemma, we produce a certified module that is the union of the refined kernel and address space library:

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} : T_{AS-PMAP}(\Psi_{AS}^{kernel}) \cup \Psi_{PMAP}^{as}$$

Applying this process to all the modules over all refinements, we link all parts of the code, except `init` certified over $\mathcal{M}_{HW}$. For readability, we hide chains of refinements. For example, T_{AS-HW} is actually $T_{AS-PMAP} \circ T_{PMAP-ALE} \circ T_{ALE-PE} \circ T_{PE-HW}$.

$$\begin{aligned}\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} :\\ T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PMAP-HW}(\Psi_{PMAP}^{as}) \cup T_{ALE-HW}(\Psi_{ALE}^{pt}) \cup\\ T_{PE-HW}(\Psi_{PE}^{mem}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit})\end{aligned}$$

To get the initialization to link with the refined module, we must make sure that the stubs that we have developed for init are compatible with the refined specifications of the actual functions. This means that we prove the following:

$$\mathtt{a}_{HW}^{kernel-init} \supseteq T_{AS-HW}(\Psi_{AS}^{kernel})(\texttt{kernel-init})$$

$$\mathtt{a}_{HW}^{meminit} \supseteq T_{PD-HW}(\Psi_{PD}^{meminit})(\texttt{mem-init}) \qquad \mathtt{a}_{HW}^{ptinit} \supseteq T_{ALD-HW}(\Psi_{ALD}^{ptinit})(\texttt{pt-init})$$

Using these properties, we apply stub strengthening to the init module:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \vdash \mathbb{C}^{init} : \Psi_{HW}^{init}$$

This certification is now linkable to the rest of the VMM and kernel, to produce the final result that we need:

$$\begin{aligned}\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} \cup \mathbb{C}^{init} :\\ T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PMAP-HW}(\Psi_{PMAP}^{as}) \cup T_{ALE-HW}(\Psi_{ALE}^{pt}) \cup\\ T_{PE-HW}(\Psi_{PE}^{mem}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \cup \Psi_{HW}^{init}\end{aligned}$$

This result means that given a certified kernel in the AS model, we can refine it to the HW model of memory by linking it with VMM implementation. Furthermore, it is safe to start this kernel by calling the `init` function, which will perform the setup, and then call the `kernel-init` function, the entry point of the high-level kernel.

6 Coq Implementation

All portions of this system have been implemented in the Coq Proof Assistant[5]. The portions of the implementation directly related to the BabyVMM verification, including C machines, refinements, specs, and related proofs (excluding frameworks) took about 3 person-months to verify. The approximate line counts for unoptimized proof are:

- Verification and refinement framework - 3000 lines
- Memory models - 200-400 lines each
- `repr` and compatibility between models - 200-400 lines each
- Compatibility of stubs and implementation - 200-400 lines per procedure
- Code verification - less than 200 lines per procedure (half of it boilerplate).

7 Related Work and Conclusion

The work presented here is a continuation of the work on Hoare-logic frameworks for verification of system software. The verification framework evolved from SCAP[8] and GCAP[3]. Although our framework does not mention separation logic[17], information hiding[16], and local action[4] explicitly, these methods had great influence on the design of the meta-language and the refinements. The definition of `repr` generalizes the work on certified garbage collector[15] to fit our concept of refinement. The project's motivation is the modular and reusable certification of the CertiKOS kernel[10].

The well-known work in OS verification is L4.verified[12, 6], which has shown a complete verification of an OS kernel. Their methodology is different, but they have considered verification of virtual memory[13, 14]. However, their current kernel verification does not abstract virtual memory, maintaining only the invariant that allows the kernel to function, and leaving the details to the user level.

The Verisoft project [9, 2, 1, 11, 18] is the work that is closest to ours. We both aim for pervasive verification of OS by doing foundational verification of all components. Both works utilize multiple machines, and require linking. As both projects aim for certification of a kernel, both have to handle virtual memory. Although Verisoft uses multiple machine models, they use them sparingly. For example, the entire microkernel, excluding assembly code, is specified in a single layer, with correctness shown as a single simulation theorem between the concurrent user thread model (CVM) and the instruction set. The authors mention that the proof of correctness is a more complex part of Verisoft. Such monolithic approach is susceptible to local modifications, where a small change in one part of microkernel may require changes to the entire proof.

Our method for verification defines many more layers, with smaller refinement proofs between them, and composes them to produce larger abstractions, ensuring that the verification is more reusable and modular. Our new framework enables us to create abstraction layers with less overhead, reducing the biggest obstacle to our approach. We have demonstrated the practicality of our approach by certifying BabyVMM, a small virtual memory manager running on simplified hardware, using a new layer for every non-trivial abstraction we could find.

Acknowledgements. We thank anonymous referees for suggestions and comments on an earlier version of this paper. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0910670 and 1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: OS Verification*, 42:389–454, 2009.
2. E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. *Proc. TACAS'08*, pages 109–123, 2008.
3. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. PLDI'07*, pages 66–77, New York, NY, USA, 2007. ACM.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. LICS'07*, pages 366–378, July 2007.
5. Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.
6. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. HoTOS'07*, San Diego, CA, USA, May 2007.
7. X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. PLDI'08*, pages 170–182. ACM, 2008.
8. X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, June 2006.
9. M. Gargano, M. A. Hillebrand, D. Leinenbach, and W. J. Paul. On the correctness of operating system kernels. In *TPHOLs'05*, 2005.
10. L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proc. APSys'11*. ACM, 2011.
11. T. In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, Computer Science Department, Nov. 2009.
12. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. SOSP'09*, pages 207–220, 2009.
13. G. Klein and H. Tuch. Towards verified virtual memory in l4. In *TPHOLs Emerging Trends '04*, Park City, Utah, USA, Sept. 2004.
14. R. Kolanski and G. Klein. Mapped separation logic. In *Proc. VSTTE'08*, pages 15–29, 2008.
15. A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI'07*, pages 468–479, 2007.
16. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, Jan. 2004.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.
18. A. Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Computer Science Department, Mar. 2010.
19. A. Vaynberg and Z. Shao. Compositional verification of BabyVMM (extended version and Coq proof). Technical Report YALEU/DCS/TR-1463, Yale University, Oct. 2012. `http://flint.cs.yale.edu/publications/babyvmm.html`.

Quantitative Reasoning for Proving Lock-Freedom

Jan Hoffmann
Yale University

Michael Marmar
Yale University

Zhong Shao
Yale University

***Abstract*—This article describes a novel quantitative proof technique for the modular and local verification of lock-freedom. In contrast to proofs based on temporal rely-guarantee requirements, this new quantitative reasoning method can be directly integrated in modern program logics that are designed for the verification of safety properties. Using a single formalism for verifying memory safety and lock-freedom allows a combined correctness proof that verifies both properties simultaneously.**

This article presents one possible formalization of this quantitative proof technique by developing a variant of concurrent separation logic (CSL) for total correctness. To enable quantitative reasoning, CSL is extended with a predicate for affine tokens to account for, and provide an upper bound on the number of loop iterations in a program. Lock-freedom is then reduced to total-correctness proofs. Quantitative reasoning is demonstrated in detail, both informally and formally, by verifying the lock-freedom of Treiber's non-blocking stack. Furthermore, it is shown how the technique is used to verify the lock-freedom of more advanced shared-memory data structures that use elimination-backoff schemes and hazard-pointers.

I. Introduction

The efficient use of multicore and multiprocessor systems requires high performance shared-memory data structures. Performance issues with traditional lock-based synchronization has generated increasing interest in *non-blocking shared-memory data structures*. In many scenarios, non-blocking data structures outperform their lock-based counterparts [1], [2]. However, their optimistic approach to concurrency complicates reasoning about their correctness.

A non-blocking data structure should guarantee that any sequence of concurrent operations that modify or access the data structure do so in a consistent way. Such a guarantee is a safety property which is implied by linearizability [3]. Additionally, a non-blocking data structure should guarantee certain *liveness properties*, which ensure that desired events eventually occur when the program is executed, independent of thread contention or the whims of the scheduler. These properties are ensured by *progress conditions* such as obstruction-freedom, lock-freedom, and wait-freedom [4], [5] (see §II). In general, it is easier to implement the data structure efficiently if the progress guarantees it makes are weaker. Lock-freedom has proven to be a sweet spot that provides a strong progress guarantee and allows for elegant and efficient implementations in practice [6], [7], [8], [9].

The formal verification of practical lock-free data structures is an interesting problem because of their relevance and the challenges they bear for current verification techniques: They employ fine-grained concurrency, shared-memory pointer-based data structures, pointer manipulation, and control flow that depends on shared state.

Classically, verification of lock-freedom is reduced to model-checking liveness properties on whole-program execution traces [10], [11], [12]. Recently, Gotsman et al. [13] have argued that lock-freedom can be reduced to modular, thread-local termination proofs of concurrent programs in which each thread only executes a single data-structure operation. Termination is then proven using a combination of concurrent separation logic (CSL) [14] and temporal trace-based rely-guarantee reasoning. In this way, proving lock-freedom is reduced to a finite number of termination proofs which can be automatically found. However, as we show in §II, this method is not intended to be applied to some lock-free data structures that are used in practice.

These temporal-logic based proofs of lock-freedom are quite different from informal lock-freedom proofs of shared data structures in the systems literature (e.g., [7], [9]). The informal argument is that the failure to make progress by a thread is always caused by *successful progress* in an operation executed by another thread. In this article, we show that this intuitive reasoning can be turned into a formal proof of lock-freedom. To this end, we introduce a *quantitative compensation scheme* in which a thread that successfully makes progress in an operation has to logically provide resources to other threads to compensate for possible interference it may have caused. Proving that all operations of a data structure adhere to such a compensation scheme is a safety property which can be formalized using minor extensions of modern program logics for fine-grained concurrent programs [14], [15], [16], [17].

We formalize one such extension in this article using CSL. We chose CSL because it has a relatively simple meta-theory and can elegantly deal with many challenges arising in the verification of concurrent, pointer-manipulating programs. Parkinson et al. [18] have shown that CSL can be used to derive modular and local safety proofs of non-blocking data structures. The key to these proofs is the identification of a *global resource invariant* on the shared-data structure that is maintained by each atomic command. However, this technique only applies to safety properties and the authors state that they are "investigating adding liveness rules to separation logic to capture properties such as obstruction/lock/wait-freedom".

We show that it is not necessary to add "liveness rules" to CSL to verify lock-freedom. As in Atkey's separation logic for quantitative reasoning [19] we extend CSL with a predicate for affine tokens to account for, and provide an upper bound on the number of loop iterations in a program. In this way, we obtain the first separation logic for total correctness of concurrent programs.

Strengthening the result of Gotsman et al. [13], we first show that lock-freedom can be reduced to the total correctness of concurrent programs in which each thread executes a finite number of data-structure operations. We then prove the total correctness of these programs using our new quantitative reasoning technique and a *quantitative resource invariant* in the sense of CSL. Thus the proof of the liveness property of being lock-free is reduced to the proof of a stronger safety property. The resulting proofs are simple extensions of memory-safety proofs in CSL and only use standard techniques such as auxiliary variables [20] and read permissions [21].

We demonstrate the practicality of our compensation-based quantitative method by verifying the lock-freedom of Treiber's non-blocking stack (§VI). We further show that the technique applies to many lock-free data structures by discussing the verification of more complex shared-memory data structures such as Michael and Scott's non-blocking queue [7], Hendler et al.'s non-blocking stack with elimination backoff [9], and Michael's non-blocking hazard-pointer data structures [8] (§VII).

Our method is a clean and intuitive modular verification technique that works correctly for shared-memory data structures that have access to thread IDs or the total number of threads in the system (see §II for details). It can not only be applied to verify total correctness but also to directly prove liveness properties or to verify termination-sensitive contextual refinement. Automation of proofs in concurrent separation logic is an orthogonal issue which is out of the scope of this paper. It would require the automatic generation of loop invariants and resource invariants. Assuming that they are in place, the automation of the proofs can rely on backward reasoning and linear programming as described by Atkey [19].

In summary, we make the following *contributions*.

1) We introduce a new compensation-based quantitative reasoning technique for proving lock-freedom of non-blocking data structures. (§III and §V)
2) We formalize our technique using an novel extension of CSL for total correctness and prove the soundness of this logic. (§IV, §V, and §VI)
3) We demonstrate the effectiveness of our approach by verifying the lock-freedom of Treiber's non-blocking stack (§VI), Michael and Scott's lock-free queue, Hendler et al.'s lock-free stack with elimination backoff, and Michael's lock-free hazard-pointer stack.

In §VII, we discuss how quantitative reasoning can verify the lock-freedom of data structures such as maps and sets, that contain loops that depend on the size of data structures. Finally, in §IX, we discuss other possible applications of quantitative reasoning for proving liveness properties including wait-freedom and starvation-freedom. Appendix II of this article contains all rules of the logic, the semantics, and the full soundness proof.

II. Non-Blocking Synchronization

Recent years have seen increasing interest in *non-blocking data structures* [1], [2]: shared-memory data structures that provide operations that are synchronized without using locks and mutual exclusion in favor of performance. A non-blocking data structure is often considered to be correct if its operations are *linearizable* [3]. Alternatively, correctness can be ensured by an invariant that is maintained by each instruction of the operations. Such an invariant is a safety property that can be proved by modern separation logics for reasoning about concurrent programs [18].

Progress Properties: In this article, we focus on complementary *liveness properties* that guarantee the *progress* of the operations of the data structure. There are three different progress properties for non-blocking data structures considered in literature. To define these, assume there is a fixed but arbitrary number of threads that are (repeatedly) accessing a shared-memory data structure exclusively via the operations it provides. Choose now a point in the execution in which one or more operations has started.

- A *wait-free* implementation guarantees that every thread can complete any started operation of the data structure in a finite number of steps [4].
- A *lock-free* implementation guarantees that some thread will complete an operation in a finite number of steps [4].
- An *obstruction-free* implementation guarantees progress for any thread that eventually executes in isolation [5] (i.e., without other active threads in the system).

Note that these definitions do not make any assumptions on the scheduler. We assume however that any code that is executed between the data-structure operations terminates. If a data structure is wait-free then it is also lock-free [4]. Similarly, lock-freedom implies obstruction-freedom [5]. Wait-free data structures are desirable because they guarantee the absence of live-locks and starvation. However, wait-free data structures are often complex and inefficient. Lock-free data structures, on the other hand, often perform more efficiently in practice. They also ensure the absence of live-locks but allow starvation. Since starvation is an unlikely event in many cases, lock-free data structures are predominant in practice and we focus on them in this paper. However, our techniques apply in principle also to wait-free data structures (see §IX).

Treiber's Stack: As a concrete example we consider Treiber's non-blocking stack [6], a classic lock-free data structure. The shared data structure is a pointer S to a linked list and the operations are *push* and *pop* as given in Figure 1.

The operation *push(v)* creates a pointer *x* to a new list node containing the data *v*. Then it stores the current stack pointer *S* in a local variable *t* and sets the next pointer of the new node *x* to *t*. Finally it attempts an atomic *compare and swap* operation *CAS(&S,t,x)* to swing *S* to point to the new node *x*. If the stack pointer *S* still contains *t* then *S* is updated and *CAS* returns *true*. In this case, the do-while loop terminates and the operation is complete. If however, the stack pointer *S* has been updated by another thread so that it no longer contains *t* then *CAS* returns *false* and leaves *S* unchanged. The do-while loop then does another iteration, updating the new list node to a new value of *S*. The operation *pop* works similarly to *push(v)*. If the stack is empty ($t == NULL$) then

```
struct Node {
  value_t data;
  Node *next;
};

Node *S;

void init()
  {S = NULL;}
```

```
void push(value_t v) {
  Node *t, *x;
  x = new Node();
  x->data = v;
  do { t = S;
    x->next = t;
  } while(!CAS(&S,t,x));
}
```

```
value_t pop() {
  Node *t, *x;
  do { t = S;
    if (t == NULL)
      {return EMPTY;}
    x = t->next;
  } while(!CAS(&S,t,x));
  return t->data;
}
```

Fig. 1. An implementation of Treiber's lock-free stack as given by Gotsman et al. [13].

```
I := -1; //initialization

ping() ≜ if I == TID then { while (true) do {} }
         else { I := TID }
```

Fig. 2. A shared data structure that shows a limitation of the method of proving lock-freedom that has been introduced by Gotsman et al. [13]. For every n, the parallel execution of n *ping* operations terminates. However, the data structure is not lock-free. (It is based on an idea from James Aspnes.)

pop returns *EMPTY*. Otherwise it repeatedly tries to update the stack pointer with the successor of the top node using a do-while loop guarded by a CAS.

Treiber's stack is lock-free but not wait-free. If other threads execute infinitely many operations they could prevent the operation of a single thread from finishing. The starvation of one thread is nevertheless only possible if infinitely many operations from other threads succeed by performing a successful *CAS*. The use of do-while loops that are guarded by CAS operations is characteristic for lock-free data structures.

Lock-Freedom and Termination: Before we verify Treiber's stack, we consider lock-freedom in general. Following an approach proposed by Gotsman et al. [13], we reduce the problem of proving lock-freedom to proving termination of a certain class of programs. Let D be any shared-memory data structure with k operations $\pi_1, \ldots, \pi_k$. It has been argued [13] that D is lock-free if and only if the following program *terminates* for every $n \in \mathbb{N}$ and every $op_1, \ldots, op_n \in \{\pi_1, \ldots \pi_k\}$: $O_n = ||_{i=1,\ldots,n}\, op_i$. However, this reduction does not apply to all shared-memory data structures. Many non-blocking data structures have operations that can distinguish different callers, for instance by accessing their thread ID. A simple example is described in Figure 2. The shared data structure consists of an integer I and a single operation *ping*. If *ping* is executed twice by the same thread without interference from another thread then the second execution of *ping* will not terminate. Otherwise, each call of *ping* immediately returns. As a result, the program $||_{i=1,\ldots,n}$ *ping* terminates for every n but the data structure is not lock-free.

We are also aware of a similar example that uses the total number of threads in the system instead of thread IDs. It is in general very common to use these system properties in non-blocking data structures. Three of the five examples in our paper use thread IDs (the hazard pointer stack, the hazard pointer queue, and the elimination-backoff stack).

Consequently, we have to prove a stronger termination property to prove that a data structure is lock-free. Instead of assuming that each client only executes one operation, we assume that each client can execute finitely many operations. To this end, we define a set of programs $\mathcal{S}^n$ that sequentially execute n operations.

$$\mathcal{S}^n = \{op_1; \ldots; op_n \mid \forall i :\ op_i \in \{\pi_1, \ldots, \pi_k\}\}$$

Let $\mathcal{S} = \bigcup_{n\in\mathbb{N}} \mathcal{S}^n$. We now define the set of programs $\mathcal{P}^m$ that execute m programs in $\mathcal{S}$ in parallel.

$$\mathcal{P}^m = \{ \quad \Big|\Big| \quad s_i \mid \forall i : s_i \in \mathcal{S}\}$$
$$i=1,\ldots,m$$

Finally, we set $\mathcal{P} = \bigcup_{m\in\mathbb{N}} \mathcal{P}^m$. For proving lock-freedom, we rely on the following theorem. By allowing a fixed but arbitrary number of operations per thread we avoid the limitations of the previous approach.

Theorem 1. The data structure D with operations $\pi_1, \ldots, \pi_k$ is lock-free if and only if every program $P \in \mathcal{P}$ terminates.

Proof. Assume first that D is lock-free. Let $P \in \mathcal{P}$. We prove that P terminates by induction on the number of incomplete operations in P, that is, operations that have not yet been started or operations that have been started but have not yet completed. If no operation is incomplete then P immediately terminates. If n operations are incomplete then the scheduler has already or will start an operation by executing one of the threads. By the definition of lock-freedom, some operation will complete independently of the choices of the scheduler. So after a finite number of steps, we reach a point in which only $n-1$ incomplete operations are left. The termination argument follows by induction.

To prove the other direction, assume now that every program $P \in \mathcal{P}$ terminates. Furthermore, assume for the sake of contradiction that D is not lock-free. Then there exists some concurrent program P_∞ that only executes operations $op \in \{\pi_1, \ldots, \pi_k\}$ and an execution trace $\mathcal{T}$ of P_∞ in which some operations have started but no operation ever completes. It follows that P_∞ diverges and $\mathcal{T}$ is therefore infinite. Let n be the number of threads in P_∞ and let s_i be the sequential program that consists of all operations that have been started by thread i in the execution trace $\mathcal{T}$ in their temporal order. Then program $||_{i=1,\ldots,n}\, s_i \in \mathcal{P}^n$ can be scheduled to produce the infinite execution trace $\mathcal{T}$. This contradicts the assumption that every program in $\mathcal{P}$ terminates. □

III. Quantitative Reasoning to Prove Lock-Freedom

A key insight of our work is that for many lock-free data structures, it is possible to give an upper bound on the total number of loop iterations in the programs in $\mathcal{P}$ (§II).

To see why, note that most non-blocking operations are based on the same optimistic approach to concurrency. They

repeatedly try to access or modify a shared-memory data structure until they can complete their operation without interference by another thread. However, lock-freedom ensures that such interference is only possible if another operation successfully makes progress:

> In an operation of a lock-free data structure, the failure of a thread to make progress is always caused by successful progress in an operation executed by another thread.

This property is the basis of a novel reasoning technique that we call a *quantitative compensation scheme*. It ensures that a thread is compensated for loop iterations that are caused by progress—often the successful completion of an operation—in another thread. In return, when a thread makes progress (e.g., completes an operation), it compensates the other threads. In this way, every thread is able to "pay" for its loop iterations without being aware of the other threads or the scheduler.

Consider for example Treiber's stack and a program P_n in which every thread only executes one operation, that is, $P_n = \|_{i=1,\ldots,n}\, s_i$ and $s_i \in \{\mathit{push}, \mathit{pop}\}$. An execution of P_n never performs more than n^2 loop iterations. Using a compensation scheme, this bound can be verified in a local and modular way. Assume that each of the threads has a number of tokens at its disposal and that each loop iteration in the program costs one token. After paying for the loop iteration, the token disappears from the system. Because it is not possible to create or duplicate tokens—tokens are an *affine resource*—the number of tokens that are initially present in the system is an upper bound on the total number of loop iterations executed.

Unfortunately, the maximum number of loop iterations performed by a thread depends on the choices of the scheduler as well as the number of operations that are performed by the other threads. To still make possible local and modular reasoning, we define a compensation scheme that enables the threads to exchange tokens. Since each loop iteration in P_n is guarded by a *CAS* operation this compensation scheme can be conveniently integrated into the specification of *CAS*. To this end, we require that (logically) $n-1$ tokens have to be available to execute a *CAS*.

(a) If the *CAS* is successful then it returns *true* and (logically) 0 tokens. Thus, the executing thread loses $n-1$ tokens.
(b) If the *CAS* is unsuccessful then it returns *false* and (logically) n tokens. Thus, the executing thread gains a token that it can use to pay for its next loop iteration.

The idea behind this compensation scheme is that every thread needs n tokens to perform a data structure operation. One token is used to pay for the first loop iteration and $n-1$ tokens are available during the loop as the loop invariant. If the *CAS* operation of a thread A is successful (case (a)) then this can cause at most $n-1$ *CAS* operations in the other threads to fail. These $n-1$ failed *CAS* operations need to return one token more than they had prior to their execution (case (b)). On the other hand, the successful thread A does not need its tokens anymore since it will exit the do-while loop. Therefore the $n-1$ tokens belonging to A are passed to the other $n-1$ threads to pay for the worst-case scenario in which this update causes $n-1$ more loop iterations.

If the *CAS* operation of a thread A fails (case (b)), then some other thread successfully updated the stack (case (a)) and thus provided a token for thread A. Since A had $n-1$ tokens before the execution of the *CAS*, it has n tokens after the execution. So thread A can pay a token for the next loop iteration and maintain its loop invariant of $n-1$ available tokens.

In our example program P_n, there are n^2 many tokens in the system at the beginning of the execution. So the number of loop iterations is bounded by n^2 and the program terminates.[1] More generally, we can use the same local and modular reasoning to prove that every program with n threads such that thread i executes m_i operations performs at most $\sum_{1\le i\le n} m_i \cdot n$ loop iterations. Thread i then starts with $m_i \cdot n$ tokens.

We will show in the following that this quantitative reasoning can be directly incorporated in total correctness proofs for these programs. We use the exact same techniques (for proving safety properties [18]) to prove liveness properties; namely *concurrent separation logic*, *auxiliary variables*, and *read permissions*. The only thing we add to separation logic is the notion of a *token* or a resource following Atkey [19].

IV. Preliminary Explanations

Before we formalize the proof outlined in §III, we give a short introduction to separation logic, quantitative reasoning, and concurrent separation logic. For the reader unfamiliar with the separation logic extensions of permissions and auxiliary variables, see Appendix I and the relevant literature [20], [21]. Our full logic is defined in Appendix II.

Separation Logic: Separation logic [22], [23] is an extension of Hoare logic [24] that simplifies reasoning about shared mutable data structures and pointers. As in Hoare logic, programs are annotated with Hoare triples using predicates $P, Q, \ldots$ over program states (heap and stack). A *Hoare triple* $[P]\,C\,[Q]$ for a program C is a total-correctness specification of C that expresses the following. If C is executed in a program state that satisfies P then C safely terminates and the execution results in a state that satisfies Q.

In addition to the traditional logical connectives, predicates of separation logic are formed by logical connectives that enable local and modular reasoning about the heap. The *separating conjunction* $P * Q$ is satisfied by a program state if the heap of that state can be split in two disjoint parts such that one sub-heap satisfies P and one sub-heap satisfies Q. It enables the safe use of the frame rule

$$\frac{[P]\,C\,[Q]}{[P * R]\,C\,[Q * R]}\ (\textsc{Frame})$$

With the frame rule it is possible to specify only the part of the heap that is modified by the program C (using predicates P and Q). This specification can then be embedded in a larger proof to state that other parts of the heap are not changed (predicate R).

[1] In fact there are at most $\binom{n}{2}$ loop iterations in the worst case. However, the n^2 bound is sufficient to prove termination.

Quantitative Reasoning: Being based on the logic of bunched implications [25], separation logic treats heap cells as *linear resources* in the sense of linear logic. It is technically unproblematic to extend separation logic to reason about *affine consumable resources* too [19]. To this end, the logic is equipped with a special predicate $\Diamond$, which states the availability of one consumable resource, or *token*. The predicate is affine because it is satisfied by every state in which *one or more tokens* are available. This in contrast to a linear predicate like $E \mapsto F$ that is only satisfied by heaps H with $|\mathrm{dom}(H)| = 1$.

Using the separating conjunction $\Diamond * P$, it is straightforward to state that two or more tokens are available. We define $\Diamond^n$ to be an abbreviation for n tokens $\Diamond * \ldots * \Diamond$ that are connected by the separating conjunction $*$.

Since we use consumable resources to model the terminating behavior of programs, the semantics of the while command are extended such that a single token is consumed, if available, at the beginning of each iteration. Correspondingly, the derivation rule for while commands ensures that a single token is available for consumption on each loop iteration and thus that the loop will execute safely:

$$\frac{P \wedge B \implies P' * \Diamond \qquad I \vdash [P']\, C\, [P]}{I \vdash [P]\ \textit{while}\ B\ \textit{do}\ C\, [P \wedge \neg B]}\ (\textsc{While})$$

The loop body C must preserve the loop invariant P under the weakened precondition P'. C is then able to execute under the assumption that one token has been consumed and still restore the invariant P, thus making a token available for possible future loop iterations.

The tokens $\Diamond$ can be freely mixed with other predicates using the usual connectives of separation logic. For instance, the formula $x \mapsto 10 \vee (x \mapsto \_ * \Diamond)$ expresses that the heap-cell referred to by the variable x points to 10, or the heap-cell points to an arbitrary value and a token is available. Together with the frame rule, the tokens enable modular reasoning about quantitative resource usage.

Concurrent Separation Logic: Concurrent separation logic (CSL) is an extension of separation logic that is used to reason about concurrent programs [14]. The idea is that shared memory regions are associated with a *resource invariant*. Each atomic block that modifies the shared region can assume that the invariant holds at the beginning of its execution and must ensure that the invariant holds at the end of the atomic block.

The original presentation of CSL [14] uses *conditional critical regions (CCRs)* for shared variables. In this article, we follow Parkinson et al. [18] and assume a global shared region with one invariant so as to simplify the syntax and the logic. An extension to CCRs is possible. For predicates I,P, and Q, the judgment $I \vdash [P]\, C\, [Q]$ states that under the global resource invariant I, in a state where P holds, the execution of the concurrent program C is safe and terminates in a state that satisfies Q.

Concurrent execution of programs C_1 and C_2 is written as $C_1 \parallel C_2$. We assume that shared variables are only accessed within atomic regions using the command *atomic(C)* and that atomic blocks are not nested. An interpretation of the resource invariant I is that it specifies a part of the heap owned by the shared region. The logical rule ATOM for the command *atomic* transfers the ownership of this heap region to the executing thread.

$$\frac{emp \vdash [P * I]\, C\, [Q * I]}{I \vdash [P]\ \textit{atomic}\{C\}\, [Q]}\ (\textsc{Atom})$$

Because the *atomic* construct ensures mutual exclusion, it is safe to share I between two programs that run in parallel. Pre- and post-conditions of concurrent programs are however combined by use of the separating conjunction[2]:

$$\frac{I \vdash [P_1]\, C_1\, [Q_1] \qquad I \vdash [P_2]\, C_2\, [Q_2]}{I \vdash [P_1 * P_2]\, C_1 \parallel C_2\, [Q_1 * Q_2]}\ (\textsc{Par})$$

Most of the other rules of sequential separation logic can be used in CSL by just adding the (unmodified) resource invariant I to the rules. The invariant is only used in the rule ATOM.

A technical detail that is crucial for the soundness of the classic rule for conjunction [24] is that we require the resource invariant I to be *precise* [14] with respect to the heap [26]. This means that, for a given heap H and stack V, there is at most one sub-heap $H' \subseteq H$ such that the sate (H', V) satisfies I. All invariants we use in this article are precise. Note that precision with respect to the resource tokens $\Diamond$ is not required since they are affine and not linear entities.

V. Formalized Quantitative Reasoning

In the following, we show how quantitative concurrent separation logic can be used to formalize the quantitative compensation scheme that we exemplified with Treiber's non-blocking stack in §III. The most important rules of this logic are described in §IV. The logic is formally defined and proved sound in Appendix II.

Before we verify realistic non-blocking data structures, we describe the formalized quantitative reasoning for a simple *producer and consumer* example.

Producer and Consumer Example: In the example in Figure 3, we have a heap location B that is shared between a number of producer and consumer threads. A producer checks whether B contains the integer 0 (i.e., B is empty). If so then it updates B with a newly produced value and terminates. Otherwise, it leaves B unchanged and terminates. A consumer checks whether B contains a non-zero integer (i.e, B is non-empty). If so then it consumes the integer, sets the contents of B to zero, and loops to check if B contains a new value to consume. If B contains 0 then the consumer terminates.

If we verify this program using our quantitative separation logic then we prove that the number of tokens specified by the precondition is an upper bound on the number of loop iterations of the program. Since the number of specified tokens is always finite, we have thus proved termination.

The challenge in the proof is that the loop iterations of the operation *Consumer* depend on the scheduler and on the number of *Producer* operations that are executed in parallel. However, it is the case that a program that uses n *Consumer*

[2]We omit the variable side-conditions here for clarity. They are included in the full set of derivation rules in Appendix II.

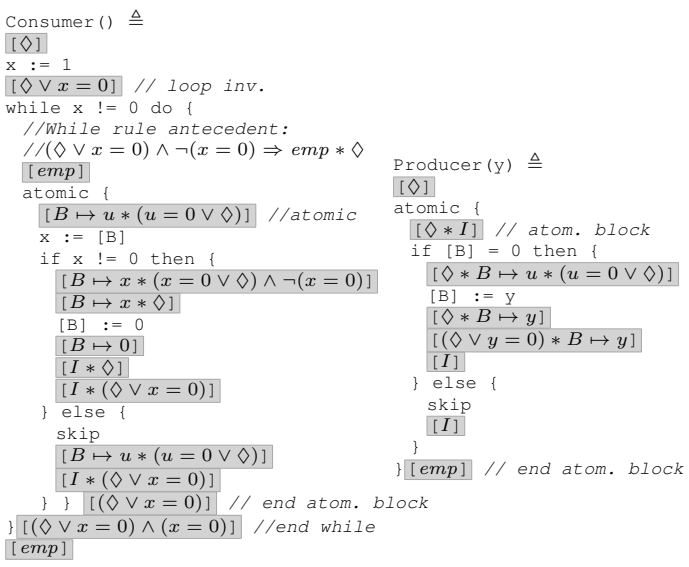


Fig. 3. A lock-free data structure B with the operations *Consumer* and *Producer*. The operation *Consumer* terminates if finitely many *Producer* operations are executed in parallel. The verification of lock-freedom and memory safety uses a compensation scheme and quantitative concurrent separation logic.

operations and m *Producer* operations performs at most $n+m$ loop iterations. We can prove this claim using our quantitative separation logic by deriving the following specifications.

$$I \vdash [\Diamond]\, \textit{Consumer()}\, [emp] \quad \text{and} \quad I \vdash [\Diamond]\, \textit{Producer(y)}\, [emp]$$

However, the modular and local specifications of these operations only hold in an environment in which all programs adhere to a certain policy. This policy can be expressed as a resource invariant I in the sense of concurrent separation logic. Intuitively, I states that the shared memory location B is read-writable, and either is empty ($B = 0$) or there is a token $\Diamond$ available. We define

$$I \triangleq \exists u.\, B \mapsto u * (u = 0 \vee \Diamond)\ .$$

Now we can read the specifications of *Consumer* and *Producer* as follows. The token $\Diamond$ in the precondition of *Consumer* is used to pay for the first loop iteration. More loop iterations are only possible if some producer updated the contents of heap location B to a non-zero integer v before the execution of the atomic block of *Consumer*. We then rely on the fact that the producer respected the resource invariant I. If $B \mapsto u$ and $u \neq 0$ then the only possibility of maintaining I is by providing a token $\Diamond$. The operation *Consumer* then updates B to zero and can thus establish the invariant I without using a token. So the token in the invariant becomes available to pay for the next loop iteration. Figure 3 contains an outline of the proof for *Producer* and *Consumer*. Note that our proof also verifies memory safety.

From Local Proofs to Lock-Freedom: Using the derived specifications of the operations and the frame rule, we inductively prove $I \vdash [\Diamond^k]\, op_1; \ldots; op_k\, [emp]$ where each op_i is a *Consumer* or *Producer* operation. In other words, we have then proved $[\Diamond^k]\, s\, [emp]$ for all $s \in \mathcal{S}^k$ (recall the definition from §II). Let now $s_i \in \mathcal{S}^{m_i}$ for $1 \leq i \leq n$. Using the rule PAR, we can then prove for $m = \sum_{i=1,\ldots,n} m_i$ that

$$I \vdash [\Diamond^m] \quad \Big|\Big|_{i=1,\ldots,n} s_i\, [emp]\ .$$

This shows that the program $||_{i=1,\ldots,n} s_i$ performs at most $m+1$ loop iterations (one token can be present in the resource invariant I) when it is executed. Following the discussion in §II, this proves that every program $p \in \mathcal{P}$ terminates and that $(B; \textit{Producer}, \textit{Consumer})$ is a lock-free data structure.

Similarly, we can in general derive a termination proof for every program in $\mathcal{P}$ from such specifications of the operations of a data structure. Assume that a shared-memory data structure $(S; \pi_1, \ldots, \pi_k)$ is given. Assume furthermore that we have verified for all $1 \leq i \leq k$ the specification $I(n) \vdash [\Diamond^{f(n)} * P]\, \pi_i\, [P]$. The notations $I(n)$ and $f(n)$ indicate that the proof can use a meta-variable n which ranges over $\mathbb{N}$. However, the proof is uniform for all n. Additionally, P might contain a variable tid for the thread ID. From this specification follows already the lock-freedom of S. To see why, we can argue as in the producer-consumer example. First, it follows for every n and $s \in \mathcal{S}^m$ that $I(n) \vdash [\Diamond^{m \cdot f(n)} * P]\, s\, [P]$. Second, a loop bound for $p = ||_{i=1,\ldots,n} s_i \in \mathcal{P}^n$ with $s_i \in \mathcal{S}^{m_i}$ is derived as follows. We use the rule PAR to prove for $m = \sum_{i=1,\ldots,n} m_i \cdot f(n)$ that

$$I(n) \vdash [\Diamond^m * \circledast_{0 \leq tid < n} P(tid)]\, p\, [\circledast_{0 \leq tid < n} P(tid)]\ .$$

Thus every $p \in \mathcal{P}$ terminates and according to the proof in §II, the data structure $(S; \pi_1, \ldots, \pi_k)$ is lock-free.

VI. LOCK-FREEDOM OF TREIBER'S STACK

We now formalize the informal proof of the lock-freedom of Treiber's stack that we described in §III. In Appendix III, we outline how the proof can be easily extended to also verify memory safety. Figure 4 shows the implementation of Treiber's stack in the while language we use in this article.

Each thread that executes *push* or *pop* operations can be in one of two states. It either has some expectation on the contents of the shared data structure S (critical state) or it does not have any expectation (non-critical state). More concretely, a thread is in a critical state if and only if it is executing a *push* or *pop* operation and is in between the two atomic blocks in the while loop. The thread then expects that $t = [S]$. The resource invariant that we will formalize in quantitative CSL can be described as follows.

> For each thread T in the system one of the following holds. (1) The thread T is in a critical state and its expectation on the shared data structure is true. (2) The thread T is in a critical state and some other thread provided T with a token. (3) The thread T is in a non-critical state.

To formalize this invariant, we have to expose the local assumption of the threads ($t = [S]$) to the global state. This is why we use auxiliary array A. If the thread with the thread ID tid is in a critical state then $A[tid]$ contains the value of its local variable t. Otherwise $A[tid]$ contains 0. Similarly, we have a second auxiliary array C such that $C[tid]$ contains a non-zero integer if and only if the thread with ID tid is in a critical state. As shown in Figure 4, the arrays A and C are

```
S := alloc(1); [S] := 0;
A := alloc(max_tid); C := alloc(max_tid);

push(v) ≜
 pushed := false;
 x := alloc(2);
 [x] := v;
 [(pushed ∨ ◇^n) * γ_r(tid,_,_)] // loop invariant
 while ( !pushed ) do {
  //While rule antecedent:
  ((pushed ∨ ◇^n) * γ_r(tid,_,_)) ∧ !pushed ⇒ ◇^{n-1} * γ_r(tid,_,_) * ◇
  [◇^{n-1} * γ_r(tid,_,_)]
  atomic {
   [◇^{n-1} * γ_r(tid,_,_) * S ↦ u * α(tid,u) * I'(tid,u)] // atom block
   [◇^{n-1} * γ(tid,_,_) * S ↦ u * I'(tid,u)] // impl. & read perm.
   t := [S];
   [◇^{n-1} * γ(tid,_,_) * (S ↦ u ∧ t = u) * I'(tid,u)] // read & frame
   C[tid] := 1
   [◇^{n-1} * γ(tid,_,1) * (S ↦ u ∧ t = u) * I'(tid,u)] // assignment
   A[tid] := t
   [◇^{n-1} * (A[tid] ↦ t ∧ t = u) * C[tid] ↦ 1 * S ↦ u * I'(tid,u)]
   [◇^{n-1} * γ_r(tid,t,1) * S ↦ u * α(tid,u) * I'(tid,u)] // perm.
   [◇^{n-1} * γ_r(tid,t,1) * I] // exist. intro & (3)
  };
  [◇^{n-1} * γ_r(tid,t,1)] // atomic block & frame
  // [x+1] := t; this is not essential for lock-freedom
  atomic {
   [◇^{n-1} * γ_r(tid,t,1) * I] // atomic block
   [◇^{n-1} * γ_r(tid,t,1) * S ↦ u * ⊛_{1≤i≤n} α(i,u)] // exist. elim.
   s := [S]; if s == t then {
    [◇^{n-1} * γ(tid,_,_) * S ↦ t * ⊛_{{1,...,n}\{tid}} (γ(i,_,_))]
    [S] := x;
    [γ(tid,_,_) * S ↦ x * I'(tid,x)] // permissions & (4)
    pushed := true
    [(pushed ∨ ◇^n) * γ(tid,_,_) * ∃u. S ↦ u * I'(tid,u)]
   } else {
    [◇^{n-1} * t ≠ u ∧ γ_r(tid,t,1) * α(tid,u) * S ↦ u * I'(tid,u)]
    [◇^n * γ(tid,t,1) * S ↦ u * I'(tid,u)] // impl. using (5)
    skip
    [(pushed ∨ ◇^n) * γ(tid,_,_) * ∃u. S ↦ u * I'(tid,u)]
   };
   C[tid] := 0
   [(pushed ∨ ◇^n) * γ(tid,_,0) * S ↦ u * I'(tid,u)]
   // write & exist. elim (above) and permissions & impl.
   [(pushed ∨ ◇^n) * γ_r(tid,_,_) * α(tid,u) * S ↦ u * I'(tid,u)]
   [(pushed ∨ ◇^n) * γ_r(tid,_,_) * I] // exist. intro
  };
  [(pushed ∨ ◇^n) * γ_r(tid,_,_)] // atomic block end
 }
```

Fig. 4. An implementation of the *push* operation of Treiber's lock-free stack in our language and the verification of the while loop. The CAS operation is implemented using an atomic block that updates the local variable *pushed*. The auxiliary array *A* contains in *A[tid]* the value of the local variable *t* of the thread with ID *tid* or zero if the thread does not assume *t = [S]*. The loop invariant $pushed \vee \Diamond^n$ states that either the new element x has been pushed to the stack S or there are n tokens available. The predicates γ and γ_r are defined in (1).

never used on the right-hand side of an assignment and are only updated in the two atomic blocks of each operation.

Let n be the number of threads in the system. We define

$$I \triangleq \exists u.\ S \mapsto u * \circledast_{0 \le i < n} \alpha(i,u)$$

$$\alpha(i,u) \triangleq \exists a, c.\ C[i] \overset{r}{\mapsto} c * A[i] \overset{r}{\mapsto} a * (c = 0 \vee a = u \vee \Diamond)$$

The resource invariant I states that the shared region has a full permission for the heap location S that points to the value u. Additionally, the predicate $\alpha(i,u)$ states for each thread i that the shared region has read permissions for $C[i]$ and $A[i]$; and that thread i is in a non-critical section ($c = 0$), that the local variable t contains the value $[S]$ ($a = u$), or that there is a token $\Diamond$ available.

We use read permissions since threads need access to the local predicate $A[tid] \overset{r}{\mapsto} t$ at some point to infer that $A[tid]$ contains the value of the local variable t. This relation of the local variable t with the array A is the only technical difficulty in the proof. Just as in safety proofs, we can now use the rules of our quantitative concurrent separation logic to verify the following Hoare triples.

$$I \vdash [\gamma_r(tid,\_,\_) * \Diamond^n]\ \ \textit{push(v)}\ \ [\gamma_r(tid,\_,\_)]$$
$$I \vdash [\gamma_r(tid,\_,\_) * \Diamond^n]\quad \textit{pop()}\quad [\gamma_r(tid,\_,\_)]$$

Where γ and γ_r are defined as:

$$\gamma(t,a,c) \triangleq A[t] \mapsto a * C[t] \mapsto c \tag{1}$$

$$\gamma_r(t,a,c) \triangleq A[t] \overset{r}{\mapsto} a * C[t] \overset{r}{\mapsto} c \tag{2}$$

Thus, the execution of any operation requires n tokens and read permission to the heap locations $A[tid]$ and $C[tid]$. After execution, the tokens are consumed and we are left with the read permissions. Figure 4 contains a proof outline for the while loop of *push*. We use the following abbreviation for parts of the invariant I that are not needed in the local proof.

$$I'(j,u) \triangleq \circledast_{i \in \{0,\ldots,n-1\}\setminus\{j\}} \alpha(i,u)$$

We have for all values u and $j \in \{0,\ldots,n-1\}$ that

$$I = \exists u.\ S \mapsto u * \alpha(j,u) * I'(j,u) \tag{3}$$

$$\Diamond^{n-1} \circledast_{i \in \{0,\ldots,n-1\}\setminus\{j\}} (\gamma_r(i,\_,\_)) \implies I'(j,u) \tag{4}$$

$$t \neq u \wedge \gamma_r(j,t,1) * \alpha(j,u) \implies \Diamond * \gamma(j,t,1) \tag{5}$$

Using these assertions, the verification of *push* and *pop* is a straightforward application of the rules of our logic. Figure 4 describes the main part—the while loop—of the proof of *push*. The loop invariant $pushed \vee \Diamond^n$ states that either the new element x has been pushed onto the stack S or there are n tokens available. In the first atomic block we leave the assumptions $I'(tid,u)$ of the other thread untouched and just establish $A[tid] \overset{r}{\mapsto} t * C[tid] \overset{r}{\mapsto} 1$.

The key aspect of the proof is the second atomic block which corresponds to the CAS operation in the original code. In the *if* case, we possibly break the assumptions of the other threads (*[S] := x*). Then we have to use $n-1$ tokens and implication (4) to re-establish $I'(tid,u)$. Since the variable *pushed* is set to *true*, we can maintain the loop invariant without using another token. In the *else* case we use the inequality $t \neq u$ and implication (5) to derive the loop invariant. Finally, we re-establish $\alpha(tid,u)$ using $C[tid] \mapsto 0$.

The verification of the while loop of *pop* is similar. By applying the proof from the end of §V to the specifications of *push* and *pop*, we have then proved the lock-freedom of Treiber's stack.

An interesting aspect of the proof is that it is not essential for a thread to know the entire resource invariant I. The only part that is needed is the implication $S \mapsto \_ * \Diamond^n * \circledast_{0 \le i < n} A[i] \overset{r}{\mapsto} \_ \implies I$. This can be used to make the assumptions $A(i)$ of the threads on the global data structure abstract. The implication $S \mapsto \_ * \Diamond^n * \circledast_{0 \le i < n} A[i] \overset{r}{\mapsto} \_ \implies$

$\exists u.\ S \mapsto \_ * \circledast_{0 \le i < n}((A[i] \overset{r}{\mapsto} \_ * \Diamond) \vee A(i,u))$ holds for all predicates $A(i,u)$. A natural candidate for such an abstraction is (concurrent) abstract predicates [27], [28]. However, such an abstraction is not needed for our goal of verifying non-blocking data structures in this paper.

VII. Advanced Lock-Free Data Structures

In this section we investigate to what extent our quantitative proof technique can be used to prove the lock-freedom of more complex shared-memory data structures.

In many cases, it is possible to derive a bound on the total number of loop iterations like we do for Treiber's stack. Table 5 gives an overview of our findings. It describes for several different non-blocking data structures the number $t(n)$ of tokens that are needed per operation in a system with n threads. The derived loop bound on a system with n threads that executes m operations is then $t(n)*m$. In the hazard-pointer data structures, the natural number ℓ is a fixed global parameter of the data structure. The details are discussed in the following.

Michael and Scott's Non-Blocking Queue: Michael and Scott's non-blocking queue [7] implements a FIFO queue using a linked list with two pointers to the head and the tail of the list. New nodes are inserted at the tail and nodes are removed from the head.

To implement the queue in a lock-free way, the insert operation can leave the data structure in an apparently inconsistent state: The new node is inserted at the tail using a CAS-guarded loop, similar to Treiber's stack. The pointer to the tail is then updated by a second CAS operation, allowing other threads to access the data structure with an inaccurate tail pointer.

To deal with this problem, the operations of the queue maintain the invariant that the tail pointer points to the last or second-to-last node during the execution and to the last node after the execution of the operation. To maintain this invariant, each CAS-guarded loop checks if the tail pointer points to a node whose next pointer is *Null*. In this case, the tail pointer is up to date and the current iteration of the while loop can continue. Otherwise, the tail pointer is updated to point to the last node of the list and the while loop is restarted.

To prove the lock-freedom of Michael and Scott's queue, we extend the invariant I that we used in the verification of Treiber's stack with an additional condition: The next pointer of the node pointed to by the tail pointer is *Null* or there is a token that can be used to pay for an additional loop iteration.

$$\exists u,t,w.\mathit{heap} \mapsto u * \mathit{tail} \mapsto t * \mathit{tail}+1 \mapsto w * \\ \circledast_{0 \le i < n} \beta(i,u,t) * (w = \mathit{nil} \vee \Diamond)$$

The formulas $\beta(i,u,t)$ are analogous to the formulas $\alpha(i,u)$ in the invariant that we used for the verification of Treiber's stack. With this invariant in a system with n threads, we can verify the operations of the queue using $n+1$ tokens in the respective preconditions.

Hazard Pointers: A limitation of Treiber's non-blocking stack is that it is only sound in the presence of garbage collection. This is due to the *ABA problem* (see for instance [8]) which appears in many algorithms that use compare-and-swap operations: Assume that a shared location which contains A is read by a thread t_1. Then thread t_2 gets activated by the scheduler, modifies the shared location to B, and then back to A. Now thread t_1 gets activated again, falsely assumes that the shared data has not been changed, and continues with its operation. The result can be a corrupted shared data structure, invalid memory access or an incorrect return value.

Data Structure	Tokens per Operation
Treiber's Stack [6]	n
Michael and Scott's Queue [7]	$n+1$
Hazard-Pointer Stack [8]	$n+(\ell \cdot n)$
Hazard-Pointer Queue [8]	$(n+1)+(\ell \cdot n)$
Elimination-Backoff Stack [9]	$n(n+1)$

Fig. 5. Quantitative reasoning for popular non-blocking data structures. The table shows the number $t(n)$ of tokens that are needed per operation in a system with n threads. The derived loop bound on a system with n threads that executes m operations is then $t(n)*m$. ℓ is a fixed global parameter of the data structure.

Michael [8] proposes *hazard pointers* to enable the safe reclamation of memory while maintaining the lock-freedom of non-blocking data structures. The idea is to introduce a global array that contains for each thread a number of hazard pointers[3] to data nodes that are currently in use by the thread. Additionally, each thread stores a local list of pointers that it wants to remove from the shared data structure (for instance by using *pop* in the case of a stack). After each successful removal of a node a thread checks if this local list has reached a fixed length threshold. If so, it checks the hazard pointers of each other thread to ensure that the pointers are not in use before reclaiming the space.

The use of hazard pointers does not affect the global resource invariants that we use in our quantitative verification technique. The reason is that hazard pointers only affect parts of the operations that are outside the loops that are guarded by CAS operations. Moreover, the worst-case number of loop iterations in this additional code can be easily determined: It is the maximal length ℓ of the local list multiplied with the maximal number of threads in the system.

For Treiber's stack with hazard pointers, the specifications of *push* and *pop* are:

$$I \vdash [\gamma_r(\mathit{tid},\_,\_) * \Diamond^n] \quad \mathit{push(v)} \quad [\gamma_r(\mathit{tid},\_,\_)]$$
$$I \vdash [\gamma_r(\mathit{tid},\_,\_) * \Diamond^{n+(\ell*n)}] \quad \mathit{pop()} \quad [\gamma_r(\mathit{tid},\_,\_)]$$

Where γ_r is defined as in (1). The resource invariant I is the same as in the specification of the version without hazard pointers.

Elimination Backoff: To improve the performance of Treiber's non-blocking stack in the presence of high contention, one can use an *elimination backoff scheme* [9]. The idea is based on the observation that a *push* operation followed by a *pop* results in a stack that is identical to the initial stack. In this case, the two operations can be *eliminated* without accessing the stack at all: The two threads use a different shared-memory

[3] In most cases, this set is just a singleton.

cell to transfer the stack element.

Our method can also be used to prove that Hendler et al's elimination-backoff stack [9] is lock-free. The main challenge in the proof is that the *push* and *pop* operations consist of two *nested* loops that are guarded by CAS operations. Assume again a system with n threads. The inner loop can be just treated as in Treiber's stack using n tokens in the precondition and 0 tokens in the postcondition. As a result, the number of tokens needed for an iteration of the outer loop is $n+1$. That means that a successful thread needs to transfer $(n-1)\cdot(n+1) = n^2-1$ tokens to the other threads to account for additional loop iterations in the other threads. Given this, we can verify the elimination-backoff stack using n^2 tokens in the precondition.

More details on the verification can be found in Appendix IV.

Non-Blocking Maps and Sets: Quantitative compensation schemes can also be used to prove the lock-freedom of non-blocking maps and sets (e.g., [29], [30]).

As in other lock-free data structures, interference in the map and set operations is only caused if the operation of another thread makes progress. For example, in the case of Harris' non-blocking linked list [29], a thread will only make an additional traversal (of the list) if there is interference caused by another thread that makes a successful traversal. The number of these additional unsuccessful traversals can be bounded using the same quantitative compensation scheme as in our previous examples.

The number of loop iterations within each list traversal depends however on the length of the list. Nevertheless, it is possible to prove an upper bound on the number of loop iterations executed by programs in $\mathcal{P}$. The reason is that each of the n threads executes a fixed number m_i of operations. Thus the total number of operations is bounded by $m = \sum_{i=1,\ldots,n} m_i$. In many important shared-memory data structures, such as lists or maps, m (or a function of m) constitutes an upper bound on the size of the shared data structure. One can then use this bound to prove an upper bound on the number of loop iterations by introducing $\Diamond^m$ in the global resource invariant. Like Atkey [19] we can use ideas from amortized resource analysis [31] to deal with variable-size data structures. By assigning tokens to each element of a data structure we derive bounds that depend on the size of the data structure without explicitly referring to its size. For instance, an inductive list-predicate that states that $k\cdot|\ell|$ tokens are available, where ℓ is the list pointed to by u can be defined as follows.

$$\begin{aligned} LSeg'(x,y,k) \Leftrightarrow & (x = y \wedge emp) \vee \\ & (\exists v, z \;\; x \mapsto v * x+1 \mapsto z * LSeg'(z,y,k) * \Diamond^k) \end{aligned}$$

VIII. Related Work

There is a large body of research on verifying safety properties and partial correctness of non-blocking data structures. See for instance [18], [32], [33] and the references therein. This work deals however with the verification of the complementary liveness property of being lock-free, which in comparison has received little attention.

Colvin and Dongol [10], [34] use manually-derived global well-founded orderings and temporal logic to prove the lock-freedom of Treiber's stack [6], Michael and Scott's queue [7], and a bounded array-based queue. Their technique is not modular but rather a whole program analysis of the most general client of the data structure. It is unclear whether the approach applies to data-structure operations with nested loops. In contrast, our method is modular, can deal with nested loops, and does not require temporal logic.

Petrank et al. [11] attempt to reduce lock-freedom to a safety property by introducing the more restrictive concept of bounded lock-freedom. It states that, in a concurrent program, there has to be progress after at most k steps, where k can depend on the input size of the program but not on the number of threads in the system. They verify bounded lock-freedom with a whole program analysis using temporal logic and the model checker Chess. The technique is demonstrated by verifying a simple concurrent program that uses Treiber's stack. Our compensation-based quantitative reasoning does not provide such an explicit bound on the steps between successful operations but rather a global bound on the number of loop iterations in the system. Additionally, our bound depends on the number of threads in the system and not on the size of the input. A conceptual difference of our work is that we prove the lock-freedom of a given data structure as opposed to the verification of a specific program. Moreover, our proofs are local and modular, and not a whole program analysis. We also show that compensation-based reasoning works for many advanced lock-free data structures.

Gotsman et al. [13] reduce lock-freedom proofs to termination proofs of programs that execute n single data structure operations in parallel. They then prove termination using separation logic and temporal rely-guarantee reasoning by layering liveness reasoning on top of a circular safety proof. Using several tools and manually formulating appropriate proof obligations, they are able to automatically verify the lock-freedom of involved algorithms such as Hendler et al.'s non-blocking stack with elimination backoff [9]. While these automation results are very impressive, the used reduction to termination is not intended to be applied to shared data structures that use thread IDs or other system information (see §II for details). In comparison, our compensation reasoning does not restrict the use of thread IDs or other system information. However, the termination proofs of [13] would also work for a modification of the reduction that we introduced in this paper.

Tofan et al. [12] describe a fully-mechanized technique based on temporal logic and rely-guarantee reasoning that is similar to the work of Gotsman et al. However, they assume weak fairness of the scheduler while we do not pose any restriction on the scheduler. Kobayashi and Sangiorgi [35] propose a type system that proves lock-freedom for programs written in the π-calculus. The target language and examples seem however to be quite different from the programs we prove lock-free in this article.

IX. Conclusion

We have shown that lock-freedom proofs of shared-memory data structures can be reduced to safety proofs in concurrent separation logic (CSL). To this end, we proposed a novel quantitative compensation scheme which can be formalized in CSL using a predicate $\Diamond$ for affine tokens. While similar logics have been used to verify the resource consumption of sequential programs [19], this is the first time that a quantitative reasoning method has been used to verify liveness properties of concurrent programs.

In the future, we plan to investigate the extent to which quantitative reasoning can be applied to other liveness properties of concurrent programs. The quantitative verification of *wait-freedom* seems to be similar to the verification of lock-freedom if we require that tokens cannot be transferred among the threads. *Obstruction-freedom* might require the creation of tokens in case of a conflict. We also plan to adapt our method to locking data structures, such as *fairness and starvation-freedom*. These properties are more challenging to verify with our quantitative method since they rely on a fair scheduler, whereas non-blocking algorithms do not. To enable such proofs, we plan to extend our compensation scheme to include the behavior of the scheduler.

Ultimately, we envision integrating our compensation-based proofs into a logic for termination-sensitive contextual refinement. We are currently developing such a logic but its description is beyond the scope of this work.

Acknowledgments

We thank James Aspnes and Alexey Gotsman for helpful discussions. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0910670 and 0915888 and 1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] W. N. Scherer III, D. Lea, and M. L. Scott, "Scalable Synchronous Queues," *Commun. ACM*, vol. 52, no. 5, pp. 100–111, 2009.

[2] N. Shavit, "Data Structures in the Multicore Age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[3] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[4] M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[5] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *23rd Int. Conf. on Distributed Comp. Systems (ICDCS'03)*, 2003.

[6] R. K. Treiber, "Systems Programming: Coping with Parallelism," IBM Almaden Research Center, Tech. Rep. RJ 5118, 1986.

[7] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *15th Symp. on Principles of Distributed Computing (PODC'96)*, 1996, pp. 267–275.

[8] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, 2004.

[9] D. Hendler, N. Shavit, and L. Yerushalmi, "A Scalable Lock-Free Stack Algorithm," in *16th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'04)*, 2004, pp. 206–215.

[10] R. Colvin and B. Dongol, "Verifying Lock-Freedom Using Well-Founded Orders," in *Theoretical Aspects of Computing - 4th International Colloquium (ICTAC'07)*, 2007, pp. 124–138.

[11] E. Petrank, M. Musuvathi, and B. Steesngaard, "Progress Guarantee for Parallel Programs via Bounded Lock-Freedom," in *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*, 2009, pp. 144–154.

[12] B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif, "Temporal Logic Verification of Lock-Freedom," in *Mathematics of Prog. Construction, 10th Int. Conf., (MPC'10)*, 2010, pp. 377–396.

[13] A. Gotsman, B. Cook, M. Parkinson, and V. Vafeiadis, "Proving that Non-Blocking Algorithms Don't Block," in *36th Symp. on Principles of Prog. Lang. (POPL'09)*, 2009, pp. 16–28.

[14] P. W. O'Hearn, "Resources, concurrency, and local reasoning," *Theor. Comput. Sci.*, vol. 375, no. 1-3, pp. 271–307, 2007.

[15] V. Vafeiadis and M. J. Parkinson, "A Marriage of Rely/Guarantee and Separation Logic," in *Concurrency Theory, 18th Int. Conference (CONCUR'07)*, 2007, pp. 256–271.

[16] X. Feng, "Local Rely-Guarantee Reasoning," in *36th Symp. on Principles of Prog. Langs. (POPL'09)*, 2009, pp. 315–327.

[17] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent Abstract Predicates," in *Object-Oriented Programming, 24th European Conf. (ECOOP'10)*, 2010, pp. 504–528.

[18] M. Parkinson, R. Bornat, and P. O'Hearn, "Modular Verification of a Non-Blocking Stack," in *34th Symp. on Principles of Prog. Lang. (POPL'07)*, 2007, pp. 297–302.

[19] R. Atkey, "Amortised Resource Analysis with Separation Logic," in *19th European Symposium on Programming (ESOP'10)*, 2010, pp. 85–103.

[20] S. S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Commun. ACM*, vol. 19, no. 5, pp. 279–285, 1976.

[21] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission Accounting in Separation Logic," in *32nd Symp. on Principles of Prog. Lang. (POPL'05)*, 2005, pp. 259–270.

[22] S. S. Ishtiaq and P. W. O'Hearn, "BI as an Assertion Language for Mutable Data Structures," in *28th Symp. on Principles of Prog. Lang. (POPL'01)*, 2001, pp. 14–26.

[23] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *17th IEEE Symp on Logic in Computer Science (LICS'02)*, 2002, pp. 55–74.

[24] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[25] P. W. O'Hearn and D. J. Pym, "The Logic of Bunched Implications," *Bulletin of Symbolic Logic*, vol. 5, no. 2, pp. 215–244, 1999.

[26] V. Vafeiadis, "Concurrent Separation Logic and Operational Semantics," *Electr. Notes Theor. Comput. Sci.*, vol. 276, pp. 335–351, 2011.

[27] M. J. Parkinson and G. M. Bierman, "Separation Logic and Abstraction," in *32nd Symp. on Principles of Prog. Lang. (POPL'05)*, 2005, pp. 247–258.

[28] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis, "Concurrent Abstract Predicates," in *Object-Oriented Programming, 24th European Conf. (ECOOP'10)*, 2010, pp. 504–528.

[29] T. L. Harris, "A Pragmatic Implementation of Non-blocking Linked-Lists," in *15th International Conf. on Distributed Computing (DISC'01)*, 2001, pp. 300–314.

[30] M. Greenwald, "Non-blocking Synchronization and System Design," Ph.D. dissertation, Stanford University, 1999, tR STAN-CS-TR-99-1624.

[31] M. Hofmann and S. Jost, "Static Prediction of Heap Space Usage for First-Order Functional Programs," in *30th Symp. on Principles of Prog. Lang. (POPL'03)*, 2003, pp. 185–197.

[32] V. Vafeiadis, "Modular Fine-Grained Concurrency Verification," Ph.D. dissertation, University of Cambridge, 2007, tR UCAM-CL-TR-726.

[33] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang, "Reasoning about Optimistic Concurrency Using a Program Logic for History," in *Concurrency Theory, 21th Int. Conference (CONCUR'10)*, 2010, pp. 388–402.

[34] R. Colvin and B. Dongol, "A General Technique for Proving Lock-Freedom," *Sci. Comput. Program.*, vol. 74, no. 3, pp. 143–165, 2009.

[35] N. Kobayashi and D. Sangiorgi, "A Hybrid Type System for Lock-Freedom of Mobile Processes," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 5, 2010.

[36] S. D. Brookes, "A Semantics for Concurrent Separation Logic," in *Concurrency Theory, 15th Int. Conference (CONCUR'04)*, 2004, pp. 16–34.

[37] P. W. O'Hearn, H. Yang, and J. C. Reynolds, "Separation and information hiding," in *31st Symp. on Principles of Prog. Lang. (POPL'04)*, 2004, pp. 268–280.

APPENDIX

I. FURTHER PRELIMINARY EXPLANATIONS

Permissions: It is sometimes necessary to share information in the form of a predicate between the invariant and a local assertion. This can be achieved in CSL by the use of permissions [21].

The predicate $E \mapsto F$ expresses that the heap location denoted by E contains the value that F denotes. Another natural reading of the predicate in the context of separation logic is that $E \mapsto F$ grants the permissions of reading from and writing to the heap location denoted by E (*permission reading*). Building upon this interpretation, read permissions state that a full read/write permission $E \mapsto F$ can be shared by two threads if the heap location denoted by E will not be modified. A full permission and two read permissions can be interchanged using the following equivalence.[4]

$$E \mapsto F \Leftrightarrow E \overset{r}{\mapsto} F * E \overset{r}{\mapsto} F$$

The two read permissions can then be shared between two threads. To write into a location, a thread needs a full permission and to read a location it only needs a read permission.

$$[x \mapsto \_]\,[x] := E\,[x \mapsto E] \text{ (WRITE)}$$

$$[E \overset{r}{\mapsto} F]\,x := [E]\,[E \overset{r}{\mapsto} F \wedge x{=}F] \text{ (READ)}$$

The remaining rules of the concurrent separation logic can remain unchanged in the presence of permissions.

Auxiliary Variables: If the rules of (concurrent) separation logic are not sufficient to prove a property of a program then we sometimes have to use *auxiliary variables* [20]. These are variables that we add to the program to monitor but not influence the computation of the original program. Thus, if we prove a property about a program using auxiliary variables then this property also holds for the program without the auxiliary variables.

More formally, we say a set Aux of variables is auxiliary for a program P if the following holds. If $x := E$ is an assignment in P and E contains a variable in Aux then $x \in Aux$. Additionally, auxiliary variables must not occur in loop or conditional tests.

II. FORMAL DEVELOPMENT AND SOUNDNESS

In the following, we give the formalization and soundness proof of our quantitative concurrent separation logic for total correctness. The proof is inspired by Vafeiadis' soundness proof [26] of concurrent separation logic and Atkey's soundness proof of his (sequential) quantitative separation logic [19]. However, we not only prove memory safety but also termination.

First, we address the syntax and semantics of our language and logic in detail. See Figure 8 for the full operational semantics of our language and Figure 9 for the Hoare-style derivation rules of the logic. The semantics are standard with

[4] A read permission is equivalent to a fractional permission with the fraction 0.5.

$$
\begin{aligned}
E &::= x \mid n \mid E + E \mid E - E \mid \dots \\
B &::= E = E \mid E < E \mid \neg B \mid B \vee B \mid \dots \\
C &::= \mathit{skip} \mid x := E \mid x := [E] \mid [E] := E \mid x := \mathit{alloc}(n) \\
&\quad \mid \mathit{dispose}(E) \mid C; C \mid C \parallel C \mid \mathit{if}\ B\ \mathit{then}\ C\ \mathit{else}\ C \\
&\quad \mid \mathit{while}\ B\ \mathit{do}\ C \mid \mathit{atomic}\ C \mid \{C\}
\end{aligned}
$$

Fig. 6. A basic while language with concurrency and dynamic allocation.

the exception of the WHILE-LOOP, WHILE-SKIP, and WHILE-ABORT rules, which deal with safe and unsafe loops in a program. Similarly, the derivation rules include an extended WHILE rule that provides a logical specification that ensures that while loops are terminating.

Language: We use a basic while language with concurrency as commonly used in the context of concurrent separation logic [36], [14], [26]. As defined in Figure 6, it is built from integer expressions E, boolean expressions B, and commands C. As in Parkinson et al. [18], we assume a global shared heap region. An extension to *conditional critical regions* [14] is possible but omitted in favor of clarity. We assume that each built-in function terminates. For simplicity, we do not include procedure calls in the language. This is an orthogonal issue that is dealt with elsewhere [27].

Semantics: Formulas and programs are interpreted with respect to a *program state* using a small-step operational semantics. Since the logic includes a consumable resource predicate, a program state consists not only of a heap and a stack but also of a natural number t which represents the number of consumable resources that are currently available to the program. To execute the body of a while loop there has to be at least 1 resource available, that is $t > 0$. After the execution of the loop body, there are $t - 1$ resources left.

Let $\mathit{Stack} = \mathit{Var} \to \mathit{Val}$ be the set of stacks and $\mathit{Heap} = \mathit{Loc} \to_{\mathit{fin}} \mathit{Val}$ be the set of heaps. Then, the set of program states is $\mathit{State} = \mathit{Heap} \times \mathit{Stack} \times \mathbb{N}$. The last component describes the number of available tokens.

The rules of the semantics are defined in Figure 8. They define an evaluation judgment of the forms

$$C, \sigma \to C', \sigma' \qquad \text{or} \qquad C, \sigma \to \bot$$

where C and C' are commands and $\sigma, \sigma' \in \mathit{State}$. Intuitively, this judgment states the following. If we execute the command C in the state σ then the next computational step results in an error ($C, \sigma \to \bot$), or it transforms the program state to σ' and execution continues with command C'. A deviation from the standard rules is in the semantics for a while loop. They ensure that a token is consumed if the body of the loop is executed. If the loop condition is satisfied and no token is available ($t = 0$) in the current state, then the result is an error.

An interesting feature of our semantics is that it does not admit infinite chains of execution steps. We prove this by defining a well-founded order $\prec$ on program states and commands. To this end, we first define the size $|C|$ of a command C as follows.

Definition 1 (Size of Commands)**.** *Let C be a command. $|C|$ is inductively defined as follows.*

$$
\begin{aligned}
|\mathit{skip}| &= 0 \\
|C_1; C_2| &= |C_1| + |C_2| + 1 \\
|C_1 \parallel C_2| &= |C_1| + |C_2| + 1 \\
|\mathit{if}\ B\ \mathit{then}\ C_1\ \mathit{else}\ C_2| &= \max(|C_1|, |C_2|) + 1 \\
|\mathit{while}\ B\ \mathit{do}\ C| &= |C| + 1 \\
|\mathit{atomic}\ C| &= |C| + 1 \\
|\{C\}| &= |C| \\
|C| &= 1 \qquad \mathit{otherwise}
\end{aligned}
$$

Definition 2 (Well-Founded Order)**.** *Let $\sigma = (H, V, t)$, $\sigma' = (H', V', t')$ be program states and let C, C' be commands. We define $(C', \sigma') \prec (C, \sigma)$ iff $t' < t$ or ($t' = t$ and $|C'| < |C|$).*

Proposition 1. *The relation $\prec$ is a well-founded order on program states.*

Lemma 1. *If $C, \sigma \to C', \sigma'$ then $(C', \sigma') \prec (C, \sigma)$.*

Proof. By inspection of the operational semantics rules. □

As a consequence of Lemma 1 and the well-foundedness of $<$ on the natural numbers, there are no infinite chains of the form $C_1, \sigma_1 \to C_2, \sigma_2 \to \cdots$.

Theorem 2. *There exist no infinite chains of the form $C_1, \sigma_1 \to C_2, \sigma_2 \to \cdots$.*

Definition 3. *For a program state σ and a command C we write $C, \sigma \Downarrow \sigma'$ if $C, \sigma \to^* \mathit{skip}, \sigma'$. Similarly, we write $C, \sigma \Downarrow \bot$ if $C, \sigma \to^* \bot$.*

An inspection of the rules of the operational semantics shows that each terminal state has the form skip, σ.

Definition 4 (Termination)**.** *We say that a program C terminates from an initial state σ if not $C, \sigma \Downarrow \bot$.*

Because our semantics do not allow infinite evaluation chains, we relate this definition to a usual small-step semantics without tokens. Since this relation is not important for the formal development we keep the discussion short. This semantic judgement is of the form $C, \tau \Rightarrow C', \tau'$ or $C, \tau \Rightarrow \bot$, where C, C' are commands and $\tau, \tau' \in \mathit{Heap} \times \mathit{Stack}$. The rules of the semantics are identical to the rules of our quantitative semantics with the token component removed. The only exceptions are the rules for while loops which are replaced by the following rules.

$$\frac{[\![B]\!](V)}{\{\mathit{while}\ B\ \mathit{do}\ C\}, (H, V) \Rightarrow \{C; \mathit{while}\ B\ \mathit{do}\ C\}, (H, V)}\ \text{(W-LOOP)}$$

$$\frac{\neg[\![B]\!](V)}{\{\mathit{while}\ B\ \mathit{do}\ C\}, \tau \Rightarrow \mathit{skip}, \tau}\ \text{(W-SKIP)}$$

Theorem 3. *Let C be a command and let $\sigma = (H, V, t)$ by a state. If not $C, \sigma \Downarrow \bot$ then there is no infinite chain of the form $C, (H, V) \Rightarrow C_1, \tau_1 \Rightarrow C_2, \tau_2 \Rightarrow \cdots$ and not $C, (H, V) \Rightarrow^* \bot$.*

$$
\begin{aligned}
\sigma \models \Diamond &\Leftrightarrow t > 0 \wedge \mathrm{dom}(H) = \emptyset \\
\sigma \models P * Q &\Leftrightarrow \exists H_1, H_2, t_1, t_2.\ H = H_1 \oplus H_2 \wedge \\
&\qquad t = t_1 + t_2 \wedge (H_1, V, t_1) \models P \wedge \\
&\qquad (H_2, V, t_2) \models Q \\
\sigma \models P \mathrel{-\!\!*} Q &\Leftrightarrow \forall H', t'.\ \text{if } H \oplus H' \text{ defined } \wedge (H', V, t') \models P \\
&\qquad \text{then } (H \oplus H', V, t + t') \models Q \\
\sigma \models E \mapsto F &\Leftrightarrow \mathrm{dom}(H) = [\![E]\!](V) \\
&\qquad \wedge H([\![E]\!](V)) = ([\![F]\!](V), \top) \\
\sigma \models E \overset{r}{\mapsto} F &\Leftrightarrow \mathrm{dom}(H) = [\![E]\!](V) \\
&\qquad \wedge H([\![E]\!](V)) = ([\![F]\!](V), r)
\end{aligned}
$$

Fig. 7. A sample of the semantics of assertions over a state $\sigma = (H, V, t)$. The semantics of the other connectives and predicates are standard.

To prove the theorem, we first prove for every $t \in \mathbb{N}$ and every program state $\tau = (H, V)$ that if $C, \tau \Rightarrow C', (H', V')$ then either $C, (H, V, t) \rightarrow C', (H', V', t')$ for some t' or $C, (H, V, t) \rightarrow \bot$. This follows immediately by an inspection of the rules. The only interesting case is the treatment of while loops for which the property is easily verified.

Given this, we see that the notion of termination $C, \sigma \Downarrow \sigma'$ corresponds exactly to the standard notion of termination under a semantics without a resource component.

Concurrent Separation Logic with Quantitative Reasoning: Following the presentation of Atkey [19], we define the predicates of quantitative separation logic as follows. Since we only deal with one resource at a time we write $\Diamond$ instead of Atkey's R.

$$
\begin{aligned}
P ::= {} & B \mid P \vee P \mid P \wedge P \mid \neg P \mid P \Rightarrow P \mid \forall x.P \mid \exists x.P \\
& \mid \Diamond \mid \mathit{emp} \mid E \mapsto E \mid E \overset{r}{\mapsto} E \mid P * P \mid P \mathrel{-\!\!*} Q \mid \circledast_{i \in I} P
\end{aligned}
$$

Following previous work [21], [26], we model assertions in the logic with *permission heaps*. Heap locations are instrumented with a permission in $\{r, \top\}$ where r is read-only and $\top$ is full permission. Permission heaps can be added using the $\oplus$ operator, which adds permissions where they overlap (and are both r), and takes the disjoint union elsewhere. The operational semantics is independent of the permissions. So we define it for heaps without permissions, which can be derived from permission heaps by deleting the permission component. Figure 7 contains the semantics of the most interesting connectives and predicates.

The rules of the program logic are given in Figure 9

Soundness: In keeping with the presentation given in [26], we define satisfaction of Hoare triples according to the inductively defined predicate $\mathsf{safe}_n(C, \sigma, I, Q)$ which states that command C will execute safely for up to n steps starting in state σ under resource invariant I and if it terminates, the resulting state will satisfy Q.

Definition 5 (Safety)**.** *For any state $\sigma = (H, V, t)$, command C, and predicates I, Q:*

- $\mathsf{safe}_0(C, \sigma, I, Q)$ *holds.*
- $\mathsf{safe}_{n+1}(C, \sigma, I, Q)$ *holds when all of the following are true:*
  1) *If $C =$ skip then $\sigma \models Q$.*
  2) *For all $t_I \in \mathbb{N}$ and all $H_I, H_F \in$ Heap such that $(H_I, V, t_I) \models I$ and $H \oplus H_I \oplus H_F$ is defined, $C, (H \oplus H_I \oplus H_F, V, t + t_I) \not\rightarrow \bot$.*
  3) *For all $t_I, t' \in \mathbb{N}$, $H_I, H_F, H' \in$ Heap, and $V' \in$ Stack such that $(H_I, V, t_I) \models I$ and $H \oplus H_I \oplus H_F$ is defined, if $C, (H \oplus H_I \oplus H_F, V, t + t_I) \rightarrow C', (H', V', t')$, then there exist H'', H_I' and t'' such that $H' = H'' \oplus H_I' \oplus H_F$, $t'' \leq t'$, $(H_I', V', t'') \models I$ and $\mathsf{safe}_n(C', (H'', V', t' - t''), I, Q)$.*

When $n > 0$, the first condition specifies that if the execution is in a terminal state, then that state satisfies the postcondition Q. The second condition states that the execution will not go wrong. The third condition ensures that each step preserves the resource invariant I, and that after executing one step, the resulting program is safe for another $n-1$ steps. In the second and third conditions, H_I and t_I represent the resources required to satisfy the global invariant I. H_F represents additional heap cells which may be needed by other parts of the program. Note that we do not include a frame t_F of consumable resources. Since predicate satisfaction is monotonic with respect to consumable resources, we do not need to distinguish between consumable resources in the shared region (t_I) and those in the frame. Also, since the operational semantics only work on concrete heaps, in condition (3) H_F will necessarily contain any heap locations that an executing thread has read permission to. By using the same H_F before and after an execution step we thus ensure that a thread cannot modify a heap location unless it has full permission at that location.

Given this, we say that a Hoare triple $[P]\, C\, [Q]$ is satisfiable under an invariant I if and only if for all $n \in \mathbb{N}$ and all states $\sigma \models P$, $\mathsf{safe}_n(C, \sigma, I, Q)$ holds. For a discussion of the motivations behind this particular characterisation of satisfaction, see [26].

Before we present the proof of soundness of the logic, we need to consider two aspects of the logic and how they interact with consumable resources: Permission Heaps [21] and Precise Assertions [14].

Permission Heaps: Let $Perm = \{r, \top\}$ be a permissions set, with r indicating read-only permission and $\top$ indicating full permission. Then, let $PHeap = Loc \rightarrow_{fin} Val \times Perm$ be the set of permission heaps. A permission heap $H \in PHeap$ is a finite mapping from locations to pairs of values and permissions. $Perm$ is equipped with a commutative partial operator $\oplus$ defined as $r \oplus r = \top$, and undefined otherwise.

We extend the permission operator $\oplus$ to value-permission pairs as follows:

$$
(v_1, p_1) \oplus (v_2, p_2) = \begin{cases} (v_1, \top) & \text{if } v_1 = v_2 \text{ and } p_1 = p_2 = r \\ \text{undefined} & \text{otherwise} \end{cases}
$$

$$\frac{}{\{x := E\}, (H, V, t) \to \mathit{skip}, (H, V|_{x=[\![E]\!](V)}, t)}\ (\textsc{Assign}) \qquad \frac{\ell = [\![E]\!](V) \qquad \ell \in \mathrm{dom}(H)}{\{x := [E]\}, (H, V, t) \to \mathit{skip}, (H, V|_{x=H(\ell)}, t)}\ (\textsc{Lookup})$$

$$\frac{[\![E]\!](V) \notin \mathrm{dom}(H)}{\{x := [E]\}, (H, V, t) \to \bot}\ (\textsc{Lookup-Abort}) \qquad \frac{\ell = [\![E]\!](V) \qquad \ell \in \mathrm{dom}(H)}{\{[E] := F\}, (H, V, t) \to \mathit{skip}, (H|_{\ell=[\![F]\!](V)}, V, t)}\ (\textsc{Mutate})$$

$$\frac{[\![E]\!](V) \notin \mathrm{dom}(H)}{\{[E] := F\}, (H, V, t) \to \bot}\ (\textsc{Mutate-Abort}) \qquad \frac{\forall i \in \{0, \ldots, n-1\}\ .\ \ell + i \notin \mathrm{dom}(H)}{\{x := \mathit{alloc}(n)\}, (H, V, t) \to \mathit{skip}, (H|_{\ell+0,\ldots,\ell+n-1=0}, V|_{x=\ell}, t)}\ (\textsc{Alloc})$$

$$\frac{\ell = [\![E]\!](V) \qquad \ell \in \mathrm{dom}(H)}{\mathit{dispose}(E), (H, V, t) \to \mathit{skip}, (H\backslash_\ell, V, t)}\ (\textsc{Dispose}) \qquad \frac{[\![E]\!](V) \notin \mathrm{dom}(H)}{\mathit{dispose}(E), (H, V, t) \to \bot}\ (\textsc{Dispose-Abort})$$

$$\frac{C_1, \sigma \to C_1', \sigma'}{\{C_1; C_2\}, \sigma \to \{C_1'; C_2\}, \sigma'}\ (\textsc{Seq1}) \qquad \frac{}{\{\mathit{skip}; C_2\}, \sigma \to C_2, \sigma}\ (\textsc{Seq2}) \qquad \frac{C_1, \sigma \to \bot}{\{C_1; C_2\}, \sigma \to \bot}\ (\textsc{Seq-Abort})$$

$$\frac{C_1, \sigma \to C_1', \sigma'}{\{C_1 \parallel C_2\}, \sigma \to \{C_1' \parallel C_2\}, \sigma'}\ (\textsc{Par1}) \qquad \frac{C_2, \sigma \to C_2', \sigma'}{\{C_1 \parallel C_2\}, \sigma \to \{C_1 \parallel C_2'\}, \sigma'}\ (\textsc{Par2}) \qquad \frac{}{\{\mathit{skip} \parallel \mathit{skip}\}, \sigma \to \mathit{skip}, \sigma}\ (\textsc{Par3})$$

$$\frac{C_1, \sigma \to \bot}{\{C_1 \parallel C_2\}, \sigma \to \bot}\ (\textsc{Par-Abort1}) \qquad \frac{C_2, \sigma \to \bot}{\{C_1 \parallel C_2\}, \sigma \to \bot}\ (\textsc{Par-Abort2}) \qquad \frac{[\![B]\!](V)}{\{\mathit{if}\ B\ \mathit{then}\ C_t\ \mathit{else}\ C_f\}, (H, V, t) \to C_t, (H, V, t)}\ (\textsc{If-True})$$

$$\frac{\neg[\![B]\!](V)}{\{\mathit{if}\ B\ \mathit{then}\ C_t\ \mathit{else}\ C_f\}, (H, V, t) \to C_f, (H, V, t)}\ (\textsc{If-False}) \qquad \frac{[\![B]\!](V) \qquad t > 0}{\{\mathit{while}\ B\ \mathit{do}\ C\}, (H, V, t) \to \{C; \mathit{while}\ B\ \mathit{do}\ C\}, (H, V, t-1)}\ (\textsc{While-Loop})$$

$$\frac{\neg[\![B]\!](V)}{\{\mathit{while}\ B\ \mathit{do}\ C\}, \sigma \to \mathit{skip}, \sigma}\ (\textsc{While-Skip}) \qquad \frac{[\![B]\!](V) \qquad t = 0}{\{\mathit{while}\ B\ \mathit{do}\ C\}, \sigma \to \bot}\ (\textsc{While-Abort}) \qquad \frac{C, \sigma \to^* \mathit{skip}, \sigma'}{\{\mathit{atomic}\ C\}, \sigma \to \mathit{skip}, \sigma'}\ (\textsc{Atom})$$

$$\frac{C, \sigma \to^* \bot}{\{\mathit{atomic}\ C\}, \sigma \to \bot}\ (\textsc{Atom-Abort})$$

Fig. 8. Small-step operational semantics

$$\frac{}{I \vdash [P]\ \mathit{skip}\ [P]}\ (\textsc{Skip}) \qquad \frac{x \notin \mathsf{fv}(I)}{I \vdash [P[E/x]]\ x := E\ [P]}\ (\textsc{Assign}) \qquad \frac{x \notin \mathsf{fv}(I, E, F)}{I \vdash [E \overset{r}{\mapsto} F]\ x := [E]\ [E \overset{r}{\mapsto} F \wedge x = F]}\ (\textsc{Lookup})$$

$$\frac{}{I \vdash [E \mapsto \_]\ [E] := F\ [E \mapsto F]}\ (\textsc{Mutate}) \qquad \frac{x \notin \mathsf{fv}(I)}{I \vdash [\mathit{emp}]\ x := \mathit{alloc}(n)\ [x + 0 \mapsto 0 \wedge \ldots \wedge x + n - 1 \mapsto 0]}\ (\textsc{Alloc})$$

$$\frac{}{I \vdash [E \mapsto \_]\ \mathit{dispose}(E)\ [\mathit{emp}]}\ (\textsc{Dispose}) \qquad \frac{I \vdash [P]\ C_1\ [Q] \qquad I \vdash [Q]\ C_2\ [R]}{I \vdash [P]\ C_1; C_2\ [R]}\ (\textsc{Seq})$$

$$\frac{\begin{array}{c} I \vdash [P_1]\ C_1\ [Q_1] \qquad I \vdash [P_2]\ C_2\ [Q_2] \\ \mathsf{fv}(I, P_1, C_1, Q_1) \cap \mathsf{wr}(C_2) = \emptyset \qquad \mathsf{fv}(I, P_2, C_2, Q_2) \cap \mathsf{wr}(C_1) = \emptyset \end{array}}{I \vdash [P_1 * P_2]\ C_1 \parallel C_2\ [Q_1 * Q_2]}\ (\textsc{Par}) \qquad \frac{I \vdash [P \wedge B]\ C_t\ [Q] \qquad I \vdash [P \wedge \neg B]\ C_f\ [Q]}{I \vdash [P]\ \mathit{if}\ B\ \mathit{then}\ C_t\ \mathit{else}\ C_f\ [Q]}\ (\textsc{If})$$

$$\frac{P \wedge B \implies P' * \Diamond \qquad I \vdash [P']\ C\ [P]}{I \vdash [P]\ \mathit{while}\ B\ \mathit{do}\ C\ [P \wedge \neg B]}\ (\textsc{While}) \qquad \frac{\mathit{emp} \vdash [P * I]\ C\ [Q * I]}{I \vdash [P]\ \mathit{atomic}\ C\ [Q]}\ (\textsc{Atom}) \qquad \frac{I * J \vdash [P]\ C\ [Q]}{I \vdash [P * J]\ C\ [Q * J]}\ (\textsc{Share})$$

$$\frac{I \vdash [P]\ C\ [Q] \qquad \mathsf{fv}(R) \cap \mathsf{wr}(C) = \emptyset}{I \vdash [P * R]\ C\ [Q * R]}\ (\textsc{Frame}) \qquad \frac{I \vdash [P]\ C\ [Q] \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{I \vdash [P']\ C\ [Q']}\ (\textsc{Consequence})$$

$$\frac{I \vdash [P_1]\ C\ [Q] \qquad I \vdash [P_2]\ C\ [Q]}{I \vdash [P_1 \vee P_2]\ C\ [Q]}\ (\textsc{Disjunction}) \qquad \frac{I \vdash [P]\ C\ [Q] \qquad x \notin \mathsf{fv}(C)}{I \vdash [\exists x.P]\ C\ [\exists x.Q]}\ (\textsc{Existential})$$

$$\frac{I \vdash [P]\ C\ [Q_1] \qquad I \vdash [P]\ C\ [Q_2] \qquad I\ \text{precise}}{I \vdash [P]\ C\ [Q_1 \wedge Q_2]}\ (\textsc{Conjunction})$$

Fig. 9. Derivation rules. fv gives the set of free variables in a command or predicate. wr gives the set of variables which are modified by a command.

We further extend $\oplus$ to permission heaps H_1 and H_2 that agree on the values at overlapping locations:

$$(H_1 \oplus H_2)(\ell) = \begin{cases} H_1(\ell) \oplus H_2(\ell) & \text{if } \ell \in \mathrm{dom}(H_1) \cap \mathrm{dom}(H_2) \\ H_1(\ell) & \text{if } \ell \in \mathrm{dom}(H_1) \setminus \mathrm{dom}(H_2) \\ H_2(\ell) & \text{otherwise} \end{cases}$$

Given this, we model assertions in the logic with permission heaps (see Figure 7). As in Vafeiadis [26], assertions are modeled with permission heaps but the operational semantics act on concrete heaps. To reconcile this, we consider regular heaps as a subset of permission heaps where the permission is always $\top$

$$\mathit{Heap} = \mathit{Loc} \rightarrow_{\mathit{fin}} \mathit{Val} \times \{\top\}$$

Then, for any permission heap H there exists a complementary permission heap H' for which $H \oplus H'$ is a concrete heap. Specifically, H' must contain the sub-heap of H that includes all the locations at which H has read permission. Define $read(H)$ to be such a sub-heap:

$$read(H) \triangleq \{(v,p) \mid (v,p) \in H \wedge p = r\}$$

Then,

$$\forall H \in \mathit{PHeap}.\ H \oplus read(H) \in \mathit{Heap}$$

Now consider Definition 5 of $\mathsf{safe}_n(C, \sigma, I, Q)$. In the definition, every time the small-step judgement $\rightarrow$ is invoked, the heap is $H \oplus H_I \oplus H_F$, which includes the universally quantified H_F. Thus, H_F will always include $read(H) \oplus read(H_I)$. This means that $H \oplus H_I \oplus H_F$ is a concrete heap, so the definition makes sense. Furthermore, since H_F is not modified by the step in condition (3), C cannot modify locations in the heap to which H has only read access.

Precise Assertions: As shown in [14], [26], in order for the logic to be sound, we require that the global resource invariant be precise in the CONJUNCTION rule. We define precise assertions [37], [14], [26] as follows. An assertion P is precise when it is satisfied by exactly one sub-heap of any heap.

Definition 6 (Precise Assertions)**.** *Let $V \in \mathit{Stack}$, $t \in \mathbb{N}$ and let P be an assertion. P is precise if and only if for all $H_1, H_2, H_1', H_2' \in \mathit{PHeap}$ such that $H_1 \oplus H_2$ is defined and $H_1 \oplus H_2 = H_1' \oplus H_2'$, if $(H_1, V, t) \models P$ and $(H_1', V, t) \models P$, then $H_1 = H_1'$.*

This definition does not consider our consumable resources. Since the assertion $\Diamond^k$ is satisfiable by any set of at least k resources, it is impossible for a predicate to specify an exact set of resources. Regardless, since the tokens are affine entities as we see in the following proof, the soundness of the logic is unaffected.

Before we prove the soundness of the rules, we have to prove two additional lemmas that are needed in the case of the rule WHILE.

Lemma 2. *Let $\sigma = (H, V, t)$ be a program state, I, Q, R predicates, $C_1; C_2$ a command, and n a natural number. If $\mathsf{safe}_n(C_1, \sigma, I, Q)$ and for all $m \leq n$ and σ' with $\sigma' \models Q$, $\mathsf{safe}_m(C_2, \sigma', I, R)$ then $\mathsf{safe}_n(C_1; C_2, \sigma, I, R)$.*

The proof of Lemma 2 is identical to the proof of the same lemma in (the formalized proof of) [26].

Lemma 3. *Let $\sigma = (H, V, t)$ be a program state, while B do C a command, and I, P, P' predicates such that $P \wedge B \implies P' * \Diamond$. If $[P']\, C\, [P]$ is satisfiable under I and $\sigma \models P$ then for all n, $\mathsf{safe}_n(\textit{while } B \textit{ do } C, \sigma, I, P \wedge \neg B)$.*

Proof. We prove the lemma by induction on n. The base case for $n = 0$ follows directly from the definition of safe_0.

Assume now that $\mathsf{safe}_n(\textit{while } B \textit{ do } C, \sigma, I, P \wedge \neg B)$ holds. To show $\mathsf{safe}_{n+1}(\textit{while } B \textit{ do } C, \sigma, I, P \wedge \neg B)$, we show that all three conditions in Definition 5 are satisfied:

1) We have *while B do $C \neq$ skip*, so this condition holds vacuously.
2) The only rule in the operational semantics which can derive $\{\textit{while } B \textit{ do } C\}, \sigma \rightarrow \bot$ is WHILE-ABORT. The premises of the rule are $[\![B]\!](V)$ and $t = 0$. We show that from $[\![B]\!](V)$ it follows that $t > 0$. Therefore WHILE-ABORT does not apply. Assume $[\![B]\!](V)$. Since $\sigma \models P$ we have then $\sigma \models P \wedge B$, and thus it follows from the premises that $\sigma \models P' * \Diamond$. From the semantics of $\Diamond$ we derive $t \geq 1$. This confirms condition (2).
3) Let $t_I, t' \in \mathbb{N}$, $H_I, H_F \in \mathit{PHeap}$, $H' \in \mathit{Heap}$ and $V' \in \mathit{Stack}$ such that $(H_I, V, t_I) \models I$ and $H \oplus H_I \oplus H_F$ is defined, and $C, (H \oplus H_I \oplus H_F, V, t + t_I) \rightarrow C', (H', V', t')$. Then the rules WHILE-LOOP or WHILE-SKIP have been applied. Since non of these rules modifies the heap (nor stack), $H' = H \oplus H_I \oplus H_F$, so let $H'' = H$ and $H_I' = H_I$.
   In the case of the rule WHILE-LOOP, we have $C' = C; \textit{while } B \textit{ do } C$ and $t' = t + t_I - 1$. Let now $t'' = t_I$. Then $t \geq 1$ (premise of WHILE-LOOP) and we have $t - 1 \geq 0$, so $t'' \leq t'$. It follows by construction, $(H_I', V', t'') = (H_I, V, t_I)$, which satisfies I. Moreover $(H'', V', t' - t'') \models P'$. Since $[P']\, C\, [P]$ is satisfiable under I, we have $\mathsf{safe}_n(C, (H'', V', t' - t''), I, P)$. By induction we have $\mathsf{safe}_m(\textit{while } B \textit{ do } C, \sigma', I, P \wedge \neg B)$ for all $m \leq n$ and all σ' with $\sigma' \models P$. Therefore we derive $\mathsf{safe}_n(C', (H'', V', t' - t''), I, P)$ with Lemma 2.
   In the case of the rule WHILE-SKIP, we have $C' = \textit{skip}$ and $t' = t + t_I$. Let again $t'' = t_I$. Then $t'' \leq t'$ and it follows that $(H_I', V', t'') = (H_I, V, t_I)$ satisfies I. Furthermore, $(H'', V', t' - t'') \models P$ and from the premise of the WHILE-SKIP we obtain $(H'', V', t' - t'') \models P \wedge \neg B$. Thus $\mathsf{safe}_n(\textit{skip}, (H'', V', t' - t''), I, P \wedge \neg B)$. (Condition (1) follows from the aforesaid and Conditions (2) and (3) by inspection of the evaluation rules.)

□

Theorem 4 (Partial Correctness)**.** *For any propositions I, P, Q and any command C, if $I \vdash [P]\, C\, [Q]$, then $[P]\, C\, [Q]$ is satisfiable under I.*

Proof. The proof is by structural induction over the derivation rules given in Figure 9. Since the only command which accesses the resource component of program state is the while loop, the

proof of every rule is essentially the same as in Vafeiadis [26] except for the rule WHILE. For all of the following, let $\sigma = (H, V, t) \in State$, C be a command, and I, P, Q be predicates.

While: Follows directly from Lemma 3.

Conjunction: To see that the definition of precise assertions is sufficient, we consider the CONJUNCTION rule. Let I be a precise assertion, Q_1, Q_2 be any assertions, and let C be a command. We show by induction that for any state $\sigma = (H, V, t)$ and any $n \in \mathbb{N}$, if $\mathsf{safe}_{n+1}(C, \sigma, I, Q_1)$ and $\mathsf{safe}_{n+1}(C, \sigma, I, Q_2)$ then $\mathsf{safe}_{n+1}(C, \sigma, I, Q_1 \wedge Q_2)$. Again, we confirm each condition:

1) if $C = \mathit{skip}$, then $\sigma \models Q_1$ and $\sigma \models Q_2$. Thus, $\sigma \models Q_1 \wedge Q_2$.
2) Since this condition does not depend on the post-condition, it is already verified by the assumption $\mathsf{safe}_{n+1}(C, \sigma, I, Q_1)$.
3) Let $t_I, t' \in \mathbb{N}$, $H_I, H_F \in PHeap$, $H' \in Heap$ and $V' \in Stack$ such that $(H_I, V, t_I) \models I$ and $H \oplus H_I \oplus H_F$ is defined, and assume that $C, (H \oplus H_I \oplus H_F, V, t{+}t_I) \to C', (H', V', t')$.
   By our assumption, there exist H'^1, $H_I'^1$ and t'^1 such that $H' = H'^1 \oplus H_I'^1 \oplus H_F$, $t'^1 \leq t'$, $(H_I'^1, V', t'^1) \models I$ and $\mathsf{safe}_n(C', (H'^1, V', t' - t'^1), I, Q_1)$. Likewise for H'^2, $H_I'^2$, t'^2 and Q_2.
   This implies that $H'^1 \oplus H_I'^1 = H'^2 \oplus H_I'^2$. Since I is precise, we know that $H_I'^1 = H_I'^2$, and thus $H'^1 = H'^2$. Finally, let $t'' = \mathsf{min}(t'^1, t'^2)$. Then, $t' - t''$ will be at least as large as both $t' - t'^1$ and $t' - t'^2$ and will thus be sufficient to ensure that both Q_1 and Q_2 hold if the execution terminates. We conclude that $\mathsf{safe}_n(C', (H'^1, V', t' - t'^1), I, Q_1 \wedge Q_2)$.

□

The total correctness of the logic is a direct consequence of Theorem 4 and Theorem 2.

Theorem 5 (Total Correctness)**.** *Let I, P, Q be propositions, C be a command, and σ be a program state. If $\sigma \models P * I$ and $I \vdash [P]\, C\, [Q]$ then every evaluation of C from the initial state σ terminates in state σ' with $\sigma' \models Q * I$.*

III. MEMORY SAFETY OF TREIBER'S STACK

To additionally verify memory safety, we have to add some auxiliary state and extend our resource invariant. The verification is then similar to the proofs in related work on verification of safety properties [32], [33]. However, there are synergies between the lock-freedom and the memory safety proof.

See Figure 10 for the full implementation of Treiber's stack in our while language. The crucial point in the verification of memory safety are the assignments *x := [t+1]* and *ret_val := [t]* in the method *pop*. Our goal is to ensure, using the resource invariant, that these locations are owned by the shared region. At the evaluation of each assignment there are two possible cases: Either the memory location that is read is still part of the stack S or it has been removed from the stack by another thread. To keep track of the memory locations that are pointed to by the stack, we introduce an inductive list predicate to describe the list pointed to by S. To keep track of the locations that have been removed from the stack we introduce an auxiliary variable that points to a second stack G that contains all the locations that have been removed from S. To this end, we push a node onto G after it is removed from S. That is, we replace the last atomic block in *pop* with the following code.

```
atomic {       // popped := CAS(S,t,x)
  s := [S];
  if s == t then {
    [S] := x;
    popped := true;
    g := [G];  // push t onto G
    [t+1] := g;
    [G] := t
  } else skip;
  C[tid] := false
}
```

The invariant I is then extended as follows where n is again the total number of threads and $\alpha(i, u)$ is defined as before.

$$
\begin{aligned}
I' \quad &\triangleq \quad \exists u.\, S \mapsto u * \circledast_{0 \leq i < n} \alpha(i, u) * G \mapsto v \\
&\quad\; * (\exists u', v'\ \mathit{LSeg}(u, u') * \mathit{LSeg}(v, v')) \wedge \\
&\quad\; \bigwedge_{0 \leq i < n} \beta(i, u, v) \\
\beta(i, u, v) \quad &\triangleq \quad \exists a, c.\, C[i] \overset{r}{\mapsto} c * A[i] \overset{r}{\mapsto} a \\
&\quad\; * (c = 0 \vee \mathit{LSeg}(u, a) \vee \mathit{LSeg}(v, a))
\end{aligned}
$$

The inductive list predicate LSeg is defined as usual [22] by

$$
\begin{aligned}
\mathit{LSeg}(x, y) \Leftrightarrow &(x = y \wedge \mathit{emp}) \vee \\
&(\exists v, z\ \ x \mapsto v * x + 1 \mapsto z * \mathit{LSeg}(z, y))
\end{aligned}
$$

The invariant ensures for each thread which is in the critical section that the local variable t points to a location that is used by the lists pointed to by S and G. Note that we can reuse the axillary arrays A and C in the formulas $\beta(i, u, v)$.

IV. VERIFICATION OF HENDLER ET AL'S ELIMINATION-BACKOFF STACK

To improve the performance of Treiber's non-blocking stack in the presence of high contention, one can use an *elimination backoff scheme* [9]. The idea is based on the observation that a *push* operation followed by a *pop* results in a stack that is identical to the initial stack. So, if a stack operation fails because of the interference of another thread then the executing thread does not immediately retry the operation. Instead, it checks if there is another thread that is trying to perform a complementary operation. In this case, the two operations can be *eliminated* without accessing the stack at all: The two threads use a different shared-memory cell to transfer the stack element.

Our method can also be used to prove that Hendler et al's elimination-backoff stack [9] is lock-free. The main challenge in the proof is that the *push* and *pop* operations consist of two *nested* loops that are guarded by CAS operations. Assume again a system with n threads. The inner loop can be just treated as in Treiber's stack using n tokens in the precondition and 0 tokens in the postcondition. As a result, the number of tokens needed

```
S := alloc(1);     // initialization
[S] := 0;
A := alloc(max_tid); // auxiliary arrays
C := alloc(max_tid); // initialized to 0

push(v) ≜
  pushed := false;
  x := alloc(2);
  [x] := v;
  while ( !pushed ) do {
    atomic {
      t := [S];     // expect t = [S]
      C[tid] := 1  // critical state starts
      A[tid] := t
    };
    [x+1] := t;
    atomic {       // pushed := CAS(S,t,x)
      s := [S];
      if s == t then {
        [S] := x;
        pushed := true;
      } else skip;
      C[tid] := 0  // critical state ends
    };
    consume(1)
  }
```

```
pop() ≜
  popped := false;
  while ( !popped ) do {
    atomic {
      t := [S];       // assume t = [S]
      C[tid] := 1     // critical state starts
      A[tid] := t
    };
    if t == 0 then { //empty stack
      ret_val := 0;
      popped := true
    } else {
      x = [t+1];
      ret_val := [t];
      atomic {        // popped := CAS(S,t,x)
        s := [S];
        if s == t then {
          [S] := x;
          popped := true;
        } else skip;
        C[tid] := false // critical state ends
      };
      consume(1)
    }
  };
  return := ret_val;
```

Fig. 10. A full implementation of Treiber's lock-free stack in our while language.

for an iteration of the outer loop is $n+1$. That means that a successful thread needs to transfer $(n-1)\cdot(n+1) = n^2-1$ tokens to the other threads to account for additional loop iterations in the other threads. Given this, we can verify the elimination-backoff stack using n^2 tokens in the precondition. Technically, we need an invariant of the form $I * J$, where I is an invariant like in Treiber's stack (for the inner loop) and J is like I but with every token $\Diamond$ replaced by $\Diamond^n$.

To make this reasoning more concrete, Figure 11 shows the loop structures of the push operation of Hendler et al's stack with elimination scheme in our while language. The auxiliary arrays $A1$ and $C1$ have the same purpose as in Treiber's stack: $C1[tid]$ indicates if thread tid is making an assumption on the value of the stack pointer S and $A1[tid]$ contains the value of the local variable t. They will be used to formulate the part of the global invariant that is crucial to maintain the loop invariant of the outer while loop. The inner while loop has the same structure as the outer loop since it is also guarded by a CAS operation. However, the address on which the CAS is performed is not fixed. Thus we need three additional auxiliary arrays to formulate part of the global invariant that is needed for the inner loop: $C2[tid]$ indicates whether thread tid is making an assumption on the shared state that is stored in *otherT*. The array $B2$ stores the memory address that is affected by this assumption and the array $A2$ stores what the assumption is.

The global invariant I can then be defined as follows.

$$\begin{aligned} I \;\triangleq\; & \exists u, v_1, \ldots, v_n.\; S \mapsto u * \mathop{\circledast}_{0\le i<n} (\delta(i,u) * \zeta(i)) \\ & * \Big(\mathop{\circledast}_{0\le j<m} col[j] \mapsto \_ * \mathop{\circledast}_{0\le i<n} B2[i] \overset{r}{\mapsto} \_ \wedge \\ & \bigwedge_{0\le i<n} \phi(i) \Big) \\ \delta(i,u) \;\triangleq\; & \exists a, c.\; C1[i] \overset{r}{\mapsto} c * A1[i] \overset{r}{\mapsto} a * (c{=}0 \vee a{=}u \vee \Diamond^n) \\ \zeta(i) \;\triangleq\; & \exists a, c.\; C2[i] \overset{r}{\mapsto} c * A2[i] \overset{r}{\mapsto} a * (c{=}0 \vee a{=}v_i \vee \Diamond) \\ \phi(i) \;\triangleq\; & \exists b.\; 0 \le b < m * B2[i] \overset{r}{\mapsto} b * col[b] \mapsto v_i \end{aligned}$$

The formulas $\delta(i,u)$ are similar to the formulas $\alpha(i,u)$ in the invariant that we used to verify Treiber's stack. However, the single token $\Diamond$ is replaced by n tokens $\Diamond^n$. The formulas $\zeta(i)$ are based on the same idea but are a bit more complicated since it is dynamically decided to which memory cell the assumption of thread i applies (namely, $col[b]$ contains the value that is stored in the thread-local variable $otherT$). The formulas $\phi(i)$ form an invariant that relates the variables v_i to the value stored in $col[B2[i]]$. The loop invariant of the outer loop is

$$\begin{aligned} & (pushed \vee \Diamond^{n\cdot n}) * A1[tid] \overset{r}{\mapsto} \_ * C1[tid] \overset{r}{\mapsto} \_ \\ & * A2[tid] \overset{r}{\mapsto} \_ * B2[tid] \overset{r}{\mapsto} \_ * C2[tid] \overset{r}{\mapsto} \_ \end{aligned}$$

The loop invariant of the inner loop is

$$(matched \vee \Diamond^n) * A2[tid] \overset{r}{\mapsto} \_ * B2[tid] \overset{r}{\mapsto} \_ * C2[tid] \overset{r}{\mapsto} \_ .$$

The proof is similar to the proof of Treiber's stack.

```
S := alloc(1);              // initialization
[S] := 0;
col := alloc(...);          // elimination array

A1 := alloc(max_tid);       // auxiliary arrays
C1 := alloc(max_tid);       // initialized to 0
A2 := alloc(max_tid);
B2 := alloc(max_tid);
C2 := alloc(max_tid);

push(v) ≜
  pushed := false;
  // ...
  while ( !pushed ) do {
    atomic {
      t := [S];               // expect t = [S]
      C1[tid] := 1;           // critical state 1 starts
      A1[tid] := t
    };
    // ...
    atomic {                  // pushed := CAS(S,t,x)
      s := [S]; if s == t then {
        [S] := x;
        pushed := true;
      } else skip;
      C1[tid] := 0            // critical state 1 ends
    };
    if !pushed then {         // elimination scheme
       // ...
       atomic {
         pos = GetPosition(...);
         B2[tid] := pos;
       }
       matched := false;
       while ( !matched ) do {
         atomic {
           otherT := col[pos]; // expectation
           C2[tid] := 1;       // critical state 2 starts
           A2[tid] := otherT
         };
         // ...
         atomic {
// pushed := CAS(col+pos,otherT,tid)
           c := col[pos]; if c == otherT then {
             col[pos] := tid;
             matched := true;
           } else skip;
           C2[tid] := 0        // critical state 2 ends
         };
       }
       // ... } }
```

Fig. 11. The loop-structure of the *push* operation of Hendler et al's stack with elimination backoff scheme [9].

Characterizing Progress Properties of Concurrent Objects via Contextual Refinements

Hongjin Liang[1,2], Jan Hoffmann[2], Xinyu Feng[1], and Zhong Shao[2]

[1] University of Science and Technology of China
[2] Yale University

Abstract. Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, or deadlock-freedom. Conventional informal or semi-formal definitions of these progress properties describe conditions under which a method call is guaranteed to complete, but it is unclear how these definitions can be utilized to formally verify system software in a layered and modular way.
In this paper, we propose a unified framework based on contextual refinements to show exactly how progress properties affect the behaviors of client programs. We give formal operational definitions of all common progress properties and prove that for linearizable objects, each progress property is equivalent to a specific type of contextual refinement that preserves termination. The equivalence ensures that verification of such a contextual refinement for a concurrent object guarantees both linearizability and the corresponding progress property. Contextual refinement also enables us to verify safety and liveness properties of client programs at a high abstraction level by soundly replacing concrete method implementations with abstract atomic operations.

1 Introduction

A concurrent object consists of shared data and a set of methods that provide an interface for client threads to manipulate and access the shared data. The synchronization of simultaneous data access within the object affects the progress of the execution of the client threads in the system.

Various progress properties have been proposed for concurrent objects. The most important ones are wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" [9].

Nevertheless, the common informal or semi-formal definitions of the progress properties are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients. In a modular verification of client threads, the concrete implementation Π of the object methods should be replaced by an abstraction (or specification) Π_A that consists of

equivalent atomic methods. The progress properties should then characterize whether and how the behaviors of a client program will be affected if a client uses Π instead of Π_A. In particular, we are interested in systematically studying whether the termination of a client using the abstract methods Π_A will be preserved when using an implementation Π with some progress guarantee.

Previous work on verifying the *safety* of concurrent objects (*e.g.*, [4, 12]) has shown that linearizability—a standard safety criterion for concurrent objects—and contextual refinement are equivalent. Informally, an implementation Π is a contextual refinement of a (more abstract) implementation Π_A, if every observable behavior of any client program using Π can also be observed when the client uses Π_A instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (*i.e.*, non-termination). Recently, Gotsman and Yang [6] showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. It is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

This paper studies all five commonly used progress properties and their relationships to contextual refinements. We propose a unified framework in which a certain type of termination-sensitive contextual refinement is equivalent to linearizability together with one of the progress properties. The idea is to identify different observable behaviors for different progress properties. For example, for the contextual refinement for lock-freedom we observe the divergence of the whole program, while for wait-freedom we also need to observe which threads in the program diverge. For lock-based progress properties, *e.g.*, starvation-freedom and deadlock-freedom, we have to take fair schedulers into account.

Our paper makes the following new contributions:

- We formalize the definitions of the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. Our formulation is based on possibly infinite event traces that are operationally generated by any client using the object.
- Based on our formalization, we prove relationships between the progress properties. For example, wait-freedom implies lock-freedom and starvation-freedom implies deadlock-freedom. These relationships form a lattice shown in Figure 1 (where the arrows represent implications). We close the lattice with a bottom element that we call *sequential termination*, a progress property in the sequential setting. It is weaker than any other progress property.
- We develop a unified framework to characterize progress properties via contextual refinements. With linearizability, each progress property is proved equivalent to a contextual refinement which takes into account divergence of programs. A companion TR [14] contains the formal proofs of our results.

By extending earlier equivalence results on linearizability [4], our contextual refinement framework can serve as a new alternative definition for the full correctness properties of concurrent objects. The contextual refinement implied by

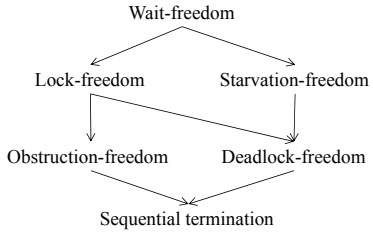


Fig. 1: Relationships between Progress Properties

linearizability and a progress guarantee precisely characterizes the properties at the abstract level that are preserved by the object implementation. When proving these properties of a client of the object, we can soundly replace the concrete method implementations by its abstract operations. On the other hand, since the contextual refinement also implies linearizability and the progress property, we can potentially borrow ideas from existing proof methods for contextual refinements, such as simulations (*e.g.*, [13]) and logical relations (*e.g.*, [2]), to verify linearizability and the progress guarantee together.

In the remainder of this paper, we first informally explain our framework in Section 2. We then introduce the formal setting in Section 3; including the definition of linearizability as the safety criterion of objects. We formulate the progress properties in Section 4 and the contextual refinement framework in Section 5. We discuss related work and conclude in Section 6.

2 Informal Account

In this section, we informally describe our results. We first give an overview of linearizability and its equivalence to the basic contextual refinement. Then we explain the progress properties and summarize our new equivalence results.

Linearizability and Contextual Refinement. *Linearizability* is a standard safety criterion for concurrent objects [9]. Intuitively, linearizability describes atomic behaviors of object implementations. It requires that each method call should appear to take effect instantaneously at some moment between its invocation and return.

Linearizability intuitively establishes a correspondence between the object implementation Π and the intended atomic operations Π_A. This correspondence can also be understood as a *contextual refinement*. Informally, we say that Π is a contextual refinement of Π_A, $\Pi \sqsubseteq \Pi_A$, if substituting Π for Π_A in any context (*i.e.*, in a client program) does not add observable behaviors. External observers cannot tell that Π_A has been replaced by Π from monitoring the behaviors of the client program.

It has been proved [4, 12] that linearizability is equivalent to a contextual refinement in which the observable behaviors are finite traces of I/O events. Thus

this basic contextual refinement can be used to distinguish linearizable objects from non-linearizable ones. But it cannot characterize progress properties of objects.

Progress Properties. Figure 2 shows several implementations of a counter with different progress guarantees that we study in this paper. A counter object provides the two methods `inc` and `dec` for incrementing and decrementing a shared variable `x`. The implementations given here are not intended to be practical but merely to demonstrate the meanings of the progress properties. We assume that every command is executed atomically.

Informally, an object implementation is *wait-free*, if it guarantees that every thread can complete any started operation of the data structure in a finite number of steps [7]. Figure 2(a) shows an ideal wait-free implementation in which the increment and the decrement are done atomically. This implementation is obviously wait-free since it guarantees termination of every method call regardless of interference from other threads. Note that realistic implementations of wait-free counters are more complex and involve arrays and atomic snapshots [1].

Lock-freedom is similar to wait-freedom but only guarantees that *some* thread will complete an operation in a finite number of steps [7]. Typical lock-free implementations (such as the well-known Treiber stack, HSY elimination-backoff stack and Harris-Michael lock-free list) use the atomic compare-and-swap instruction `cas` in a loop to repeatedly attempt an update until it succeeds. Figure 2(b) shows such an implementation of the counter object. It is lock-free, because whenever `inc` and `dec` operations are executed concurrently, there always exists some successful update. Note that this object is not wait-free. For the following program (2.1), the `cas` instruction in the method called by the left thread may continuously fail due to the continuous updates of `x` made by the right thread.

```
inc();   ||   while(true) inc();          (2.1)
```

Herlihy *et al.* [8] propose *obstruction-freedom* which "guarantees progress for any thread that eventually executes in isolation" (*i.e.*, without other active threads in the system). They present two double-ended queues as examples. In Figure 2(c) we show an obstruction-free counter that may look contrived but nevertheless illustrates the idea of the progress property.

The implementation introduces a variable `i`, and lets `inc` perform the atomic increment after increasing `i` to 10 and `dec` do the atomic decrement after decreasing `i` to 0. Whenever a method is executed in isolation (*i.e.*, without interference from other threads), it will complete. Thus the object is obstruction-free. It is not lock-free, because for the client

```
inc();   ||   dec();                      (2.2)
```

which executes an increment and a decrement concurrently, it is possible that neither of the method calls returns. For instance, under a specific schedule, every increment over `i` made by the left thread is immediately followed by a decrement from the right thread.

```
1 inc() { x := x + 1; }
2 dec() { x := x - 1; }
```

(a) Wait-Free (Ideal) Impl.

```
1 inc() {
2  local t, b;
3  do {
4    t := x;
5    b := cas(&x,t,t+1);
6  } while(!b);
7 }
```

(b) Lock-Free Impl.

```
 1 inc() {
 2   while (i < 10) {
 3     i := i + 1;
 4   }
 5   x := x + 1;
 6 }
 7 dec() {
 8   while (i > 0) {
 9     i := i - 1;
10   }
11   x := x - 1;
12 }
```

(c) Obstruction-Free Impl.

```
1 inc() {
2  TestAndSet_lock();
3  x := x + 1;
4  TestAndSet_unlock();
5 }
```

(d) Deadlock-Free Impl.

```
 1  inc() {
 2    Bakery_lock();
 3    x := x + 1;
 4    Bakery_unlock();
 5  }
```

(e) Starvation-Free Impl.

Fig. 2: Counter Objects with Methods `inc` and `dec`

Wait-freedom, lock-freedom, and obstruction-freedom are progress properties for non-blocking implementations, where a delay of a thread cannot prevent other threads from making progress. In contrast, deadlock-freedom and starvation-freedom are progress properties for lock-based implementations. A delay of a thread holding a lock will block other threads which request the lock.

Deadlock-freedom and starvation-freedom are often defined in terms of locks and critical sections. Deadlock-freedom guarantees that some thread will succeed in acquiring the lock, and starvation-freedom states that every thread attempting to acquire the lock will eventually succeed [9]. For example, a test-and-set spin lock is deadlock-free but not starvation-free. In a concurrent access, some thread will successfully set the bit and get the lock, but there might be a thread that is continuously failing to get the lock. Lamport's bakery lock is starvation-free. It ensures that threads can acquire locks in the order of their requests.

However, as noted by Herlihy and Shavit [10], the above definitions based on locks are unsatisfactory, because it is often difficult to identify a particular field in the object as a lock. Instead, they suggest defining them in terms of method calls. They also notice that the above definitions implicitly assume that every thread acquiring the lock will eventually release it. This assumption requires *fair* scheduling, *i.e.*, every thread gets eventually executed.

Following Herlihy and Shavit [10], we say an object is *deadlock-free*, if in each *fair* execution there always exists some method call that can finish. As an example in Figure 2(d), we use a test-and-set lock to synchronize the increments of the counter. Since some thread is guaranteed to acquire the test-and-set lock, the method call of that thread is guaranteed to finish. Thus the object is deadlock-free. Similarly, a *starvation-free* object guarantees that every method call can finish in fair executions. Figure 2(e) shows a counter implemented with Lamport's bakery lock. It is starvation-free since the bakery lock ensures that every thread can acquire the lock and hence every method call can eventually complete.

	Wait-Free	Lock-Free	Obstruction-Free	Deadlock-Free	Starvation-Free
Π_A	(t, Div.)	Div.	Div.	Div.	(t, Div.)
Π	(t, Div.)	Div.	Div. if Isolating	Div. if Fair	(t, Div.) if Fair

Table 1: Characterizing Progress Properties via Contextual Refinements $\Pi \sqsubseteq \Pi_A$

Our Results. None of the above definitions of the five progress properties describes their guarantees regarding the behaviors of client code. In this paper, we define several contextual refinements to characterize the effects over client behaviors when the client uses objects with some progress properties. We show that linearizability together with a progress property is equivalent to a certain termination-sensitive contextual refinement. Table 1 summarizes our results.

For each progress property, the new contextual refinement $\Pi \sqsubseteq \Pi_A$ is defined with respect to a divergence behavior and/or a specific scheduling at the implementation level (the third row in Table 1) and at the abstract side (the second row), in addition to the I/O events in the basic contextual refinement for linearizability.

- For wait-freedom, we need to observe the divergence of each individual thread t, represented by "(t, Div.)" in Table 1, at both the concrete and the abstract levels. We show that, if the thread t of a client program diverges when the client uses a linearizable and wait-free object Π, then thread t must also diverge when using Π_A instead.
- The case for lock-freedom is similar, except that we now consider the divergence behaviors of the *whole* client program rather than individual threads (denoted by "Div." in Table 1). If a client diverges when using a linearizable and lock-free object Π, it must also diverge when it uses Π_A instead.
- For obstruction-freedom, we consider the behaviors of *isolating* executions at the concrete side (denoted by "Div. if Isolating" in Table 1). In those executions, eventually only one thread is running. We show that, if a client diverges in an isolating execution when it uses a linearizable and obstruction-free object Π, it must also diverge in some abstract execution.
- For deadlock-freedom, we only care about *fair* executions at the concrete level (denoted by "Div. if Fair" in Table 1).
- For starvation-freedom, we observe the divergence of each individual thread at both levels and restrict our considerations to fair executions for the concrete side ("(t, Div.) if Fair" in Table 1). Any thread using Π can diverge in a fair execution, only if it also diverges in some abstract execution.

These new contextual refinements can characterize linearizable objects with progress properties. We will formalize the results and give examples in Section 5.

3 Formal Setting and Linearizability

In this section, we formalize linearizability and show its equivalence to a contextual refinement that preserves safety properties only. This equivalence is the basis of our new results that relate progress properties and contextual refinements.

$$
\begin{array}{rrcl}
(\mathit{Expr}) & E & ::= & \ldots \quad (\mathit{BExp})\ B ::= \ldots \quad (\mathit{Instr})\ c ::= \mathbf{print}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \mathbf{end} \\
 & & \mid & \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \| \ldots \| C \\
(\mathit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

Fig. 3: Syntax of the Programming Language

$$
\begin{array}{rrcl}
(\mathit{State}) & \mathcal{S} & ::= & \ldots \qquad (\mathit{ThrdID})\ \mathsf{t} \in \mathit{Nat} \\
(\mathit{Evt}) & e & ::= & (\mathsf{t}, f, n) \mid (\mathsf{t}, \mathbf{ret}, n) \mid (\mathsf{t}, \mathbf{obj}) \mid (\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \\
 & & \mid & (\mathsf{t}, \mathbf{out}, n) \mid (\mathsf{t}, \mathbf{clt}) \mid (\mathsf{t}, \mathbf{clt}, \mathbf{abort}) \mid (\mathsf{t}, \mathbf{term}) \mid (\mathbf{spawn}, n) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T \quad \text{(co-inductive)}
\end{array}
$$

Fig. 4: States and Event Traces

Language and Semantics. We use a similar language as in previous work of Liang and Feng [12]. As shown in Figure 3, a program W consists of several client threads that run in parallel. Each thread could call the methods declared in the object Π. A method f is defined as a pair (x, C), where x is the formal argument and C is the method body. The object Π could be either concrete with fine-grained code that we want to verify, or abstract (usually denoted as Π_A in the following) that we consider as the specification. For the latter case, each method body should be an atomic operation of the form $\langle C \rangle$ and it should be always safe to execute it. For simplicity, we assume there is only one object in the program W and each method takes one argument only.

Most commands are standard. Clients can use $\mathbf{print}(E)$ to produce observable external events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls. To discuss progress properties later, we introduce an auxiliary command **end**. It is a special marker that can be added at the end of a thread, but is not supposed to be used directly by programmers. The **skip** statement plays two roles here: a statement that has no computation effects or a flag to show the end of an execution.

We use $\mathcal{S}$ for a program state. Program transitions $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$ generate events e defined in Figure 4. A method invocation event (t, f, n) is produced when thread t executes $x := f(E)$, where n is the value of the argument E. A return $(\mathsf{t}, \mathbf{ret}, n)$ is produced with the return value n. $\mathbf{print}(E)$ generates an output $(\mathsf{t}, \mathbf{out}, n)$, and **end** generates a termination marker $(\mathsf{t}, \mathbf{term})$. Other steps generate either normal object actions $(\mathsf{t}, \mathbf{obj})$ (for steps inside method calls) or silent client actions $(\mathsf{t}, \mathbf{clt})$ (for client steps other than $\mathbf{print}(E)$). For transitions leading to the error state **abort** (*e.g.*, invalid memory access), fault events are produced: $(\mathsf{t}, \mathbf{obj}, \mathbf{abort})$ by the object method code and $(\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ by the client code. We also introduce an auxiliary event $(\mathbf{spawn}, n)$, saying that n threads are spawned. It will be useful later when defining fair scheduling (in Section 4). We write $\mathsf{tid}(e)$ for the thread ID in the event e. The predicate $\mathsf{is\_clt}(e)$ states that the event e is either a silent client action, an output, or a client fault. We write $\mathsf{is\_inv}(e)$ and $\mathsf{is\_ret}(e)$ to denote that e is a method invocation

$$\mathcal{T}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{T \mid \exists W', \mathcal{S}'.\ (W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}') \vee (W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} \mathbf{abort}\}$$
$$\mathcal{H}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_hist}(T) \mid T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}[\![W,\mathcal{S}]\!]\}$$

Fig. 5: Generation of Finite Event Traces

and a return, respectively. The predicate $\mathsf{is\_abt}(e)$ denotes a fault of the object or the client. Method invocations, returns and object faults are called *history* events, which will be used to define linearizability below. Outputs, client faults and object faults are called *observable* events.

An event trace T is a finite or infinite sequence of events. We write $T(i)$ for the i-th event of T. $\mathsf{last}(T)$ is the last event in a finite T. The trace $T(1..i)$ is the sub-trace $T(1), \ldots, T(i)$ of T, and $|T|$ is the length of T ($|T| = \omega$ if T is infinite). The trace $T|_{\mathsf{t}}$ represents the sub-trace of T consisting of all events whose thread ID is t. We can use $\mathsf{get\_hist}(T)$ to project T to the sub-trace consisting of all the history events, and $\mathsf{get\_obsv}(T)$ for the sub-trace of all the observable events. Finite traces of history events are called *histories*.

In Figure 5, we define $\mathcal{T}[\![W,\mathcal{S}]\!]$ for the prefix-closed set of finite traces produced by the executions of $(W,\mathcal{S})$. We use $(W,\mathcal{S}) \stackrel{T}{\longmapsto}{}^{*} (W',\mathcal{S}')$ for zero or multiple-step program transitions that generate the trace T. We also define $\mathcal{H}[\![W,\mathcal{S}]\!]$ and $\mathcal{O}[\![W,\mathcal{S}]\!]$ to get histories and finite observable traces produced by the executions of $(W,\mathcal{S})$. The TR [14] contains more details about the language.

Linearizability and Basic Contextual Refinement. We formulate linearizability following its standard definition [11]. Below we sketch the basic concepts. Detailed formal definitions can be found in the companion TR [14].

Linearizability is defined using histories. We say a return e_2 *matches* an invocation e_1, denoted as $\mathsf{match}(e_1, e_2)$, iff they have the same thread ID. An invocation is *pending* in T if no matching return follows it. We can use $\mathsf{pend\_inv}(T)$ to get the set of pending invocations in T. We handle pending invocations in a history T in the standard way [11]: we append zero or more return events to T, and drop the remaining pending invocations. The result is denoted by $\mathsf{completions}(T)$. It is a set of histories, and for each history in it, every invocation has a matching return event.

Definition 1 (Linearizable Histories). $T \preceq_{\mathsf{lin}} T'$ *iff*

1. $\forall \mathsf{t}.\ T|_{\mathsf{t}} = T'|_{\mathsf{t}}$;
2. *there exists a bijection* $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T'|\}$ *such that* $\forall i.\ T(i) = T'(\pi(i))$ *and* $\forall i, j.\ i < j \wedge \mathsf{is\_ret}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j)$.

That is, T is linearizable *w.r.t.* T' if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls. Then an *object* is linearizable iff each of its concurrent histories after completions is linearizable *w.r.t.* some *legal sequential* history. We use $\Pi_A \rhd (\mathcal{S}_a, T')$ to mean that T' is a legal sequential history generated by any client using the specification Π_A with an abstract initial state $\mathcal{S}_a$.

Definition 2 (Linearizability of Objects). *The object's implementation Π is linearizable w.r.t. Π_A under a refinement mapping φ, denoted by $\Pi \preceq_\varphi \Pi_A$, iff*

$\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a, T.\ T \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}]\!] \wedge (\varphi(\mathcal{S}) = \mathcal{S}_a)$
$\Longrightarrow\ \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\mathcal{S}_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'$.

Here the partial mapping φ: State$\rightharpoonup$ State relates concrete states to abstract ones.

The side condition $\varphi(\mathcal{S}) = \mathcal{S}_a$ in the above definition requires the initial concrete state $\mathcal{S}$ to be well-formed in that it represents a valid abstract state $\mathcal{S}_a$. For instance, φ may need $\mathcal{S}$ to contain a linked list and relate it to an abstract mathematical set in $\mathcal{S}_a$ for a set object. Besides, φ should always require the client states in $\mathcal{S}$ and $\mathcal{S}_a$ to be identical.

Next we define a contextual refinement between the concrete object and its specification, which is equivalent to linearizability.

Definition 3 (Basic Contextual Refinement). *$\Pi \sqsubseteq_\varphi \Pi_A$ iff*

$\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a)$
$\Longrightarrow\ \mathcal{O}[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}]\!] \subseteq \mathcal{O}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]$.

Remember that $\mathcal{O}[\![W, \mathcal{S}]\!]$ represents the prefix-closed set of observable event traces generated during the executions of $(W, \mathcal{S})$, which is defined in Figure 5.

Following Filipović *et al.* [4], we can prove that linearizability is equivalent to this contextual refinement. We give the proofs in the TR [14].

Theorem 1 (Basic Equivalence). *$\Pi \preceq_\varphi \Pi_A \iff \Pi \sqsubseteq_\varphi \Pi_A$.*

Theorem 1 allows us to use $\Pi \sqsubseteq_\varphi \Pi_A$ to identify linearizable objects. However, we cannot use it to characterize progress properties of objects. For the following example, $\Pi \sqsubseteq_\varphi \Pi_A$ holds although no concrete method call of `f` could finish (we assume this object contains a method `f` only).

Π(`f`) : `while(true) skip;` Π_A(`f`) : `skip;` C : `print(1); f(); print(1);`

The reason is that $\Pi \sqsubseteq_\varphi \Pi_A$ considers a *prefix-closed* set of event traces at the abstract side. For the above client C, the observable behaviors of **let** Π **in** C can all be found in the prefix-closed set of behaviors produced by **let** Π_A **in** C.

4 Formalizing Progress Properties

We define progress in Figure 6 as properties over both event traces T and object implementations Π. We say an object implementation Π has a progress property P iff all its event traces have the property. Here we use $\mathcal{T}_\omega$ to generate the event traces. Its definition in Figure 6 is similar to $\mathcal{T}[\![W, \mathcal{S}]\!]$ of Figure 5, but $\mathcal{T}_\omega[\![W, \mathcal{S}]\!]$ is for the set of finite or infinite event traces produced by *complete* executions. We use $(W, \mathcal{S}) \xmapsto{T}{}^{\omega} \cdot$ to denote the existence of a T-labelled infinite execution. $(W, \mathcal{S}) \xmapsto{T}{}^{*} (\textbf{skip}, \_)$ represents a terminating execution that produces T. By using $\lfloor W \rfloor$, we append **end** at the end of each thread to explicitly mark the termination of the thread. We also insert the spawning event $(\textbf{spawn}, n)$ at the

Definition. An object Π satisfies P under a refinement mapping φ, $P_\varphi(\Pi)$, iff $\forall n, C_1, \ldots, C_n, \mathcal{S}, T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n), \mathcal{S}]\!] \wedge (\mathcal{S} \in dom(\varphi)) \implies P(T)$.

$$\mathcal{T}_\omega[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathbf{spawn}, |W|) :: T \mid (\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{\omega} \cdot \vee (\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{*} (\mathbf{skip}, \_) \vee (\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{*} \mathbf{abort}\}$$

$$\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n \rfloor \stackrel{\text{def}}{=} \mathbf{let}\ \Pi\ \mathbf{in}\ (C_1; \mathbf{end}) \| \ldots \| (C_n; \mathbf{end})$$

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}((\mathbf{spawn}, n) :: T) \stackrel{\text{def}}{=} n$$

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is\_inv}(e) \wedge \neg \exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\}$$

$$\mathsf{prog\text{-}t}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j))$$

$$\mathsf{prog\text{-}s}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))$$

$$\mathsf{abt}(T) \ \text{ iff } \ \exists i.\ \mathsf{is\_abt}(T(i))$$

$$\mathsf{sched}(T) \ \text{ iff } \ |T| = \omega \wedge \mathsf{pend\_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend\_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega$$

$$\mathsf{fair}(T) \ \text{ iff } \ |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$$

$$\mathsf{iso}(T) \ \text{ iff } \ |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})$$

$$\mathsf{wait\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{starvation\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

$$\mathsf{lock\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt} \qquad \mathsf{deadlock\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt}$$

$$\mathsf{obstruction\text{-}free} \ \text{ iff } \ \mathsf{sched} \wedge \mathsf{iso} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

Fig. 6: Formalizing Progress Properties

$$\mathsf{lock\text{-}free} \iff \mathsf{wait\text{-}free} \vee \mathsf{prog\text{-}s} \qquad \mathsf{starvation\text{-}free} \iff \mathsf{wait\text{-}free} \vee \neg\mathsf{fair}$$

$$\mathsf{obstruction\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{iso} \qquad \mathsf{deadlock\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{fair}$$

Fig. 7: Relationships between Progress Properties

beginning of T, where n is the number of threads in W. Then we can use $\mathsf{tnum}(T)$ to get the number n, which is needed to define fairness, as shown below.

Before formulating each progress property over event traces, we first define some auxiliary properties in Figure 6. $\mathsf{prog\text{-}t}(T)$ guarantees that every method call in T eventually finishes. $\mathsf{prog\text{-}s}(T)$ guarantees that *some* pending method call finishes. Different from prog-t, the return event $T(j)$ in prog-s does not have to be a matching return of the pending invocation e. $\mathsf{abt}(T)$ says that T ends with a fault event.

There are three useful conditions on scheduling. The basic requirement for a good schedule is sched. If T is infinite and there exist pending calls, then at least one pending thread should be scheduled infinitely often. In fact, there are two possible reasons causing a method call of thread t to pend. Either t is no longer scheduled, or it is always scheduled but the method call never finishes. sched rules out the bad schedule where no thread with an invoked method is active. For instance, the following infinite trace does *not* satisfy sched.

$$(\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: \ldots$$

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (|(T|_{\mathsf{t}})| = \omega)\}$$
$$\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{i\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{iso}(T)\}$$
$$\mathcal{O}_{f\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{t\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{ft\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$

Fig. 8: Generation of Complete Event Traces

If T is infinite, $\mathsf{fair}(T)$ requires every non-terminating thread be scheduled infinitely often; and $\mathsf{iso}(T)$ requires eventually only one thread be scheduled. We can see that a fair schedule is a good schedule satisfying sched.

At the bottom of Figure 6 we define the progress properties formally. We omit the parameter T in the formulae to simplify the presentation. An event trace T is wait-free (*i.e.*, $\mathsf{wait\text{-}free}(T)$ holds) if under the good schedule sched, it guarantees prog-t unless it ends with a fault. $\mathsf{lock\text{-}free}(T)$ is similar except that it guarantees prog-s. Starvation-freedom and deadlock-freedom guarantee prog-t and prog-s under fair scheduling. Obstruction-freedom guarantees prog-t if some pending thread is always scheduled (sched) and runs in isolation (iso).

Figure 7 contains lemmas that relate progress properties. For instance, an event trace is starvation-free, iff it is wait-free or not fair. These lemmas give us the relationship lattice in Figure 1. To close the lattice, we also define a progress property in the sequential setting. *Sequential termination* guarantees that every method call must finish in a trace produced by a sequential client. The formal definition is given in the companion TR [14], and we prove that it is implied by each of the five progress properties for concurrent objects.

5 Equivalence to Contextual Refinements

We extend the basic contextual refinement in Definition 3 to observe progress as well as linearizability. For each progress property, we carefully choose the observable behaviors at the concrete and the abstract levels.

5.1 Observable Behaviors

In Figure 8, we define various observable behaviors for the termination-sensitive contextual refinements.

We use $\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!]$ to represent the set of observable event traces produced by complete executions of $(W,\mathcal{S})$. Recall that $\mathsf{get\_obsv}(T)$ gets the sub-trace of T consisting of all the observable events only. Unlike the prefix-closed set $\mathcal{O}[\![W,\mathcal{S}]\!]$, this definition utilizes $\mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]$ (see Figure 6) whose event traces are all complete and could be infinite. Thus it allows us to observe divergence of the whole program. $\mathcal{O}_{i\omega}$ and $\mathcal{O}_{f\omega}$ take the complete observable traces of *isolating* and fair executions respectively. Here $\mathsf{iso}(T)$ and $\mathsf{fair}(T)$ are defined in Figure 6.

P	wait-free	lock-free	obstruction-free	deadlock-free	starvation-free
$\Pi \sqsubseteq_\varphi^P \Pi_A$	$\mathcal{O}_{t\omega} \subseteq \mathcal{O}_{t\omega}$	$\mathcal{O}_\omega \subseteq \mathcal{O}_\omega$	$\mathcal{O}_{i\omega} \subseteq \mathcal{O}_\omega$	$\mathcal{O}_{f\omega} \subseteq \mathcal{O}_\omega$	$\mathcal{O}_{ft\omega} \subseteq \mathcal{O}_{t\omega}$

Table 2: Contextual Refinements $\Pi \sqsubseteq_\varphi^P \Pi_A$ for Progress Properties P

We could also observe divergence of individual threads rather than the whole program. We define $\mathsf{div\_tids}(T)$ to collect the set of threads that diverge in the trace T. Then we write $\mathcal{O}_{t\omega}[\![W, \mathcal{S}]\!]$ to get both the observable behaviors and the diverging threads in the complete executions. $\mathcal{O}_{ft\omega}[\![W, \mathcal{S}]\!]$ is defined similarly but considers fair executions only.

More on divergence. In general, divergence means non-termination. For example, we could say that the following two-threaded program (5.1) must diverge since it never terminates.

```
x := x + 1;    ||    while(true) skip;          (5.1)
```

But for individual threads, divergence is not equivalent to non-termination, since a non-terminating thread may either have an infinite execution or simply be not scheduled from some point due to unfair scheduling. We view only the former case as divergence. For instance, in an unfair execution, the left thread of (5.1) may never be scheduled and hence it has no chance to terminate. It does not diverge. Similarly, for the following program (5.2),

```
while(true) skip;    ||    while(true) skip;          (5.2)
```

the whole program must diverge, but it is possible that a single thread does not diverge in an execution.

5.2 New Contextual Refinements and Equivalence Results

In Table 2, we summarize the definitions of the termination-sensitive contextual refinements. Each new contextual refinement follows the basic one in Definition 3 but takes different observable behaviors as specified in Table 2. For example, the contextual refinement for wait-freedom is formally defined as follows:

$$\Pi \sqsubseteq_\varphi^{\mathsf{wait\text{-}free}} \Pi_A \;\text{ iff }\; (\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a) \Longrightarrow \mathcal{O}_{t\omega}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S})]\!] \subseteq \mathcal{O}_{t\omega}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]).$$

Theorem 2 says that linearizability with a progress property P together is equivalent to the corresponding contextual refinement $\sqsubseteq_\varphi^P$.

Theorem 2 (Equivalence). *$\Pi \preceq_\varphi \Pi_A \wedge P_\varphi(\Pi) \iff \Pi \sqsubseteq_\varphi^P \Pi_A$, where P is* wait-free*,* lock-free*,* obstruction-free*,* deadlock-free *or* starvation-free*.*

Here we assume the object specification Π_A is *total*, *i.e.*, the abstract operations never block. We provide the proofs of our equivalence results in the TR [14].

The contextual refinement for wait-freedom takes $\mathcal{O}_{t\omega}$ at both the concrete and the abstract levels. The divergence of individual threads as well as I/O events are treated as observable behaviors. The intuition of the equivalence is as

follows. Since a wait-free object Π guarantees that every method call finishes, we have to blame the client code itself for the divergence of a thread using Π. That is, even if the thread uses the abstract object Π_A, it must still diverge.

As an example, consider the client program (2.1). Intuitively, for any execution in which the client uses the abstract operations, only the right thread $\mathtt{t}_2$ diverges. Thus $\mathcal{O}_{t\omega}$ of the abstract program is a singleton set $\{(\epsilon, \{\mathtt{t}_2\})\}$. When the client uses the wait-free object in Figure 2(a), its $\mathcal{O}_{t\omega}$ set is still $\{(\epsilon, \{\mathtt{t}_2\})\}$. It does not produce more observable behaviors. But if it uses a non-wait-free object (such as the one in Figure 2(b)), the left thread $\mathtt{t}_1$ does not necessarily finish. The $\mathcal{O}_{t\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$. It produces more observable behaviors than the abstract client, breaking the contextual refinement. Thanks to observing div_tids that collects the diverging threads, we can rule out non-wait-free objects which may cause more threads to diverge.

$\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ takes coarser observable behaviors. We observe the divergence of the whole client program by using $\mathcal{O}_\omega$ at both the concrete and the abstract levels. Intuitively, a lock-free object Π ensures that some method call will finish, thus the client using Π diverges only if there are an infinite number of method calls. Then it must also diverge when using the abstract object Π_A.

For example, consider the client (2.1). The whole client program diverges in every execution both when it uses the lock-free object in Figure 2(b) and when it uses the abstract one. The $\mathcal{O}_\omega$ set of observable behaviors is $\{\epsilon\}$ at both levels. On the other hand, the following client must terminate and print out both 1 and 2 in every execution. The $\mathcal{O}_\omega$ set is $\{1::2::\epsilon, 2::1::\epsilon\}$ at both levels.

```
inc(); print(1);    ||    dec(); print(2);        (5.3)
```

Instead, if the client (5.3) uses the non-lock-free object in Figure 2(c), it may diverge and nothing is printed out. The $\mathcal{O}_\omega$ set becomes $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, which contains more behaviors than the abstract side. Thus $\Pi \sqsubseteq_{\varphi}^{\text{lock-free}} \Pi_A$ fails.

Obstruction-freedom ensures progress for isolating executions in which eventually only one thread is running. Correspondingly, $\Pi \sqsubseteq_{\varphi}^{\text{obstruction-free}} \Pi_A$ restricts our considerations to isolating executions. It takes $\mathcal{O}_{i\omega}$ at the concrete level and $\mathcal{O}_\omega$ at the abstract level.

To understand the equivalence, consider the client (5.3) again. For isolating executions with the obstruction-free object in Figure 2(c), it *must* terminate and print out both 1 and 2. The $\mathcal{O}_{i\omega}$ set at the concrete level is $\{1::2::\epsilon, 2::1::\epsilon\}$, the same as the set $\mathcal{O}_\omega$ of the abstract side. Non-obstruction-free objects in general do not guarantee progress for some isolating executions. If the client uses the object in Figure 2(d) or (e), the $\mathcal{O}_{i\omega}$ set is $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, not a subset of the abstract $\mathcal{O}_\omega$ set. The undesired empty observable trace is produced by unfair executions, where a thread acquires the lock and gets suspended and then the other thread would keep requesting the lock forever (it is executed in isolation).

$\Pi \sqsubseteq_{\varphi}^{\text{deadlock-free}} \Pi_A$ uses $\mathcal{O}_{f\omega}$ at the concrete side, ruling out undesired divergence caused by unfair scheduling. For the client (5.3) with the object in Figure 2(d) or (e), its $\mathcal{O}_{f\omega}$ set is same as the set $\mathcal{O}_\omega$ at the abstract level.

For $\Pi \sqsubseteq_{\varphi}^{\text{starvation-free}} \Pi_A$, we still consider only fair executions at the concrete level (similar to deadlock-freedom), but observe the divergence of individual

threads rather than the whole program (similar to wait-freedom). It uses $\mathcal{O}_{ft\omega}$ at the concrete side and $\mathcal{O}_{t\omega}$ at the abstract level. For the client (5.3) with the starvation-free object in Figure 2(e), no thread diverges in any fair execution. Then the set $\mathcal{O}_{ft\omega}$ of observable behaviors is $\{(1::2::\epsilon, \emptyset), (2::1::\epsilon, \emptyset)\}$, which is same as the set $\mathcal{O}_{t\omega}$ at the abstract level.

Observing threaded divergence allows us to distinguish starvation-free objects from deadlock-free objects. Consider the client (2.1). Under fair scheduling, we know only the right thread $\mathtt{t}_2$ would diverge when using the starvation-free object in Figure 2(e). The set $\mathcal{O}_{ft\omega}$ is $\{(\epsilon, \{\mathtt{t}_2\})\}$. It coincides with the abstract behaviors $\mathcal{O}_{t\omega}$. But when using the deadlock-free object of Figure 2(d), the $\mathcal{O}_{ft\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$, breaking the contextual refinement.

6 Related Work and Conclusion

There is a large body of work discussing the five progress properties and the contextual refinements individually. Our work in contrast studies their relationships, which have not been considered much before.

Gotsman and Yang [6] propose a new linearizability definition that preserves lock-freedom, and suggest a connection between lock-freedom and a termination-sensitive contextual refinement. We do not redefine linearizability here. Instead, we propose a unified framework to systematically relate all the five progress properties plus linearizability to various contextual refinements.

Herlihy and Shavit [10] informally discuss all the five progress properties. Our definitions in Section 4 mostly follow their explanations, but they are more formal and close the gap between program semantics and their history-based interpretations. We also notice that their obstruction-freedom is inappropriate for some examples (see TR [14]), and propose a different definition that is closer to the common intuition [9]. In addition, we relate the progress properties to contextual refinements, which consider the extensional effects on client behaviors.

Fossati *et al.* [5] propose a uniform approach in the π-calculus to formulate both the standard progress properties and their observational approximations. Their technical setting is completely different from ours. Also, their observational approximations for lock-freedom and wait-freedom are strictly weaker than the standard notions. Their deadlock-freedom and starvation-freedom are not formulated, and there is no observational approximation given for obstruction-freedom. In comparison, our framework relates each of the five progress properties (plus linearizablity) to an *equivalent* contextual refinement.

There are also formulations of progress properties based on temporal logics. For example, Petrank *et al.* [15] formalize the three non-blocking properties and Dongol [3] formalize all the five progress properties, using linear temporal logics. Those formulations make it easier to do model checking (*e.g.*, Petrank *et al.* [15] also build a tool to model check a variant of lock-freedom), while our contextual refinement framework is potentially helpful for modular Hoare-style verification.

Conclusion. We have introduced a contextual refinement framework to unify various progress properties. For linearizable objects, each progress property is

equivalent to a specific termination-sensitive contextual refinement, as summarized in Table 1. The framework allows us to verify safety and liveness properties of client programs at a high abstraction level by replacing concrete method implementations with abstract operations. It also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together, which we leave as future work.

Acknowledgments. We would like to thank anonymous referees for their helpful suggestions and comments. This work is supported in part by China Scholarship Council, NSFC grants 61073040 and 61229201, NCET grant NCET-2010-0984, and the Fundamental Research Funds for the Central Universities (Grant No. WK0110000018). It is also supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0915888 and 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Aspnes, J., Herlihy, M.: Wait-free data structures in the asynchronous PRAM model. In: SPAA. pp. 340–349 (1990)
2. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: CSL. pp. 107–121 (2012)
3. Dongol, B.: Formalising progress properties of non-blocking programs. In: ICFEM. pp. 284–303 (2006)
4. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010)
5. Fossati, L., Honda, K., Yoshida, N.: Intensional and extensional characterisation of global progress in the π-calculus. In: CONCUR. pp. 287–301 (2012)
6. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: ICALP. pp. 453–465 (2011)
7. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
8. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS. pp. 522–529 (2003)
9. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (Apr 2008)
10. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS. pp. 313–328 (2011)
11. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
12. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI. p. to appear (2013)
13. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL. pp. 455–468 (2012)
14. Liang, H., Hoffmann, J., Feng, X., Shao, Z.: The extended version of the present paper (2013), `http://kyhcs.ustcsz.edu.cn/relconcur/prog`
15. Petrank, E., Musuvathi, M., Steensgaard, B.: Progress guarantee for parallel programs via bounded lock-freedom. In: PLDI. pp. 144–154 (2009)

Characterizing Progress Properties of Concurrent Objects via Contextual Refinements (Extended Version)

Hongjin Liang[1,2], Jan Hoffmann[2], Xinyu Feng[1], and Zhong Shao[2]

[1] University of Science and Technology of China
[2] Yale University

Abstract. Implementations of concurrent objects should guarantee linearizability and a progress property such as wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, or deadlock-freedom. Conventional informal or semi-formal definitions of these progress properties describe conditions under which a method call is guaranteed to complete, but it is unclear how these definitions can be utilized to formally verify system software in a layered and modular way.
In this paper, we propose a unified framework based on contextual refinements to show exactly how progress properties affect the behaviors of client programs. We give formal operational definitions of all common progress properties and prove that for linearizable objects, each progress property is equivalent to a specific type of contextual refinement that preserves termination. The equivalence ensures that verification of such a contextual refinement for a concurrent object guarantees both linearizability and the corresponding progress property. Contextual refinement also enables us to verify safety and liveness properties of client programs at a high abstraction level by soundly replacing concrete method implementations with abstract atomic operations.

1 Introduction

A concurrent object consists of shared data and a set of methods that provide an interface for client threads to manipulate and access the shared data. The synchronization of simultaneous data access within the object affects the progress of the execution of the client threads in the system.

Various progress properties have been proposed for concurrent objects. The most important ones are wait-freedom, lock-freedom and obstruction-freedom for non-blocking implementations, and starvation-freedom and deadlock-freedom for lock-based implementations. These properties describe conditions under which method calls are guaranteed to successfully complete in an execution. For example, lock-freedom guarantees that "infinitely often some method call finishes in a finite number of steps" [10].

Nevertheless, the common informal or semi-formal definitions of the progress properties are difficult to use in a modular and layered program verification because they fail to describe how the progress properties affect clients. In a modular

verification of client threads, the concrete implementation Π of the object methods should be replaced by an abstraction (or specification) Π_A that consists of equivalent atomic methods. The progress properties should then characterize whether and how the behaviors of a client program will be affected if a client uses Π instead of Π_A. In particular, we are interested in systematically studying whether the termination of a client using the abstract methods Π_A will be preserved when using an implementation Π with some progress guarantee.

Previous work on verifying the *safety* of concurrent objects (*e.g.*, [4, 13]) has shown that linearizability—a standard safety criterion for concurrent objects—and contextual refinement are equivalent. Informally, an implementation Π is a contextual refinement of a (more abstract) implementation Π_A, if every observable behavior of any client program using Π can also be observed when the client uses Π_A instead. To obtain equivalence to linearizability, the observable behaviors include I/O events but not divergence (*i.e.*, non-termination). Recently, Gotsman and Yang [7] showed that a client program that diverges using a linearizable and *lock-free* object must also diverge when using the abstract operations instead. Their work reveals a connection between lock-freedom and a form of contextual refinement which preserves termination as well as safety properties. It is unclear how other progress guarantees affect termination of client programs and how they are related to contextual refinements.

This paper studies all five commonly used progress properties and their relationships to contextual refinements. We propose a unified framework in which a certain type of termination-sensitive contextual refinement is equivalent to linearizability together with one of the progress properties. The idea is to identify different observable behaviors for different progress properties. For example, for the contextual refinement for lock-freedom we observe the divergence of the whole program, while for wait-freedom we also need to observe which threads in the program diverge. For lock-based progress properties, *e.g.*, starvation-freedom and deadlock-freedom, we have to take fair schedulers into account.

Our paper makes the following new contributions:

- We formalize the definitions of the five most common progress properties: wait-freedom, lock-freedom, obstruction-freedom, starvation-freedom, and deadlock-freedom. Our formulation is based on possibly infinite event traces that are operationally generated by any client using the object.
- Based on our formalization, we prove relationships between the progress properties. For example, wait-freedom implies lock-freedom and starvation-freedom implies deadlock-freedom. These relationships form a lattice shown in Figure 1 (where the arrows represent implications). We close the lattice with a bottom element that we call *sequential termination*, a progress property in the sequential setting. It is weaker than any other progress property.
- We develop a unified framework to characterize progress properties via contextual refinements. With linearizability, each progress property is proved equivalent to a contextual refinement which takes into account divergence of programs. The formal proofs of our results can be found in Appendix B.

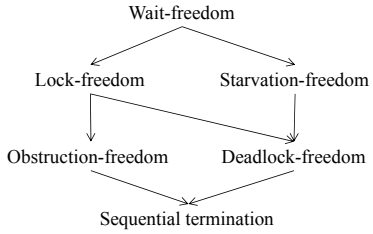


Fig. 1: Relationships between Progress Properties

By extending earlier equivalence results on linearizability [4], our contextual refinement framework can serve as a new alternative definition for the full correctness properties of concurrent objects. The contextual refinement implied by linearizability and a progress guarantee precisely characterizes the properties at the abstract level that are preserved by the object implementation. When proving these properties of a client of the object, we can soundly replace the concrete method implementations by its abstract operations. On the other hand, since the contextual refinement also implies linearizability and the progress property, we can potentially borrow ideas from existing proof methods for contextual refinements, such as simulations (*e.g.*, [14]) and logical relations (*e.g.*, [2]), to verify linearizability and the progress guarantee together.

In the remainder of this paper, we first informally explain our framework in Section 2. We then introduce the formal setting in Section 3; including the definition of linearizability as the safety criterion of objects. We formulate the progress properties in Section 4 and the contextual refinement framework in Section 5. We discuss related work and conclude in Section 6.

2 Informal Account

In this section, we informally describe our results. We first give an overview of linearizability and its equivalence to the basic contextual refinement. Then we explain the progress properties and summarize our new equivalence results.

Linearizability and Contextual Refinement. *Linearizability* is a standard safety criterion for concurrent objects [10]. Intuitively, linearizability describes atomic behaviors of object implementations. It requires that each method call should appear to take effect instantaneously at some moment between its invocation and return.

Linearizability intuitively establishes a correspondence between the object implementation Π and the intended atomic operations Π_A. This correspondence can also be understood as a *contextual refinement*. Informally, we say that Π is a contextual refinement of Π_A, $\Pi \sqsubseteq \Pi_A$, if substituting Π for Π_A in any context (*i.e.*, in a client program) does not add observable behaviors. External observers cannot tell that Π_A has been replaced by Π from monitoring the behaviors of the client program.

It has been proved [4, 13] that linearizability is equivalent to a contextual refinement in which the observable behaviors are finite traces of I/O events. Thus this basic contextual refinement can be used to distinguish linearizable objects from non-linearizable ones. But it cannot characterize progress properties of objects.

Progress Properties. Figure 2 shows several implementations of a counter with different progress guarantees that we study in this paper. A counter object provides the two methods `inc` and `dec` for incrementing and decrementing a shared variable `x`. The implementations given here are not intended to be practical but merely to demonstrate the meanings of the progress properties. We assume that every command is executed atomically.

Informally, an object implementation is *wait-free*, if it guarantees that every thread can complete any started operation of the data structure in a finite number of steps [8]. Figure 2(a) shows an ideal wait-free implementation in which the increment and the decrement are done atomically. This implementation is obviously wait-free since it guarantees termination of every method call regardless of interference from other threads. Note that realistic implementations of wait-free counters are more complex and involve arrays and atomic snapshots [1].

Lock-freedom is similar to wait-freedom but only guarantees that *some* thread will complete an operation in a finite number of steps [8]. Typical lock-free implementations (such as the well-known Treiber stack, HSY elimination-backoff stack and Harris-Michael lock-free list) use the atomic compare-and-swap instruction `cas` in a loop to repeatedly attempt an update until it succeeds. Figure 2(b) shows such an implementation of the counter object. It is lock-free, because whenever `inc` and `dec` operations are executed concurrently, there always exists some successful update. Note that this object is not wait-free. For the following program (2.1), the `cas` instruction in the method called by the left thread may continuously fail due to the continuous updates of `x` made by the right thread.

```
inc();   ||   while(true) inc();                    (2.1)
```

Herlihy *et al.* [9] propose *obstruction-freedom* which "guarantees progress for any thread that eventually executes in isolation" (*i.e.*, without other active threads in the system). They present two double-ended queues as examples. In Figure 2(c) we show an obstruction-free counter that may look contrived but nevertheless illustrates the idea of the progress property.

The implementation introduces a variable `i`, and lets `inc` perform the atomic increment after increasing `i` to 10 and `dec` do the atomic decrement after decreasing `i` to 0. Whenever a method is executed in isolation (*i.e.*, without interference from other threads), it will complete. Thus the object is obstruction-free. It is not lock-free, because for the client

```
inc();   ||   dec();                               (2.2)
```

which executes an increment and a decrement concurrently, it is possible that neither of the method calls returns. For instance, under a specific schedule, every

```
1 inc() { x := x + 1; }
2 dec() { x := x - 1; }
```

(a) Wait-Free (Ideal) Impl.

```
1 inc() {
2  local t, b;
3  do {
4    t := x;
5    b := cas(&x,t,t+1);
6  } while(!b);
7 }
```

(b) Lock-Free Impl.

```
 1 inc() {
 2   while (i < 10) {
 3     i := i + 1;
 4   }
 5   x := x + 1;
 6 }
 7 dec() {
 8   while (i > 0) {
 9     i := i - 1;
10   }
11   x := x - 1;
12 }
```

(c) Obstruction-Free Impl.

```
1 inc() {
2  TestAndSet_lock();
3  x := x + 1;
4  TestAndSet_unlock();
5 }
```

(d) Deadlock-Free Impl.

```
 1  inc() {
 2    Bakery_lock();
 3    x := x + 1;
 4    Bakery_unlock();
 5  }
```

(e) Starvation-Free Impl.

Fig. 2: Counter Objects with Methods `inc` and `dec`

increment over `i` made by the left thread is immediately followed by a decrement from the right thread.

Wait-freedom, lock-freedom, and obstruction-freedom are progress properties for non-blocking implementations, where a delay of a thread cannot prevent other threads from making progress. In contrast, deadlock-freedom and starvation-freedom are progress properties for lock-based implementations. A delay of a thread holding a lock will block other threads which request the lock.

Deadlock-freedom and starvation-freedom are often defined in terms of locks and critical sections. Deadlock-freedom guarantees that some thread will succeed in acquiring the lock, and starvation-freedom states that every thread attempting to acquire the lock will eventually succeed [10]. For example, a test-and-set spin lock is deadlock-free but not starvation-free. In a concurrent access, some thread will successfully set the bit and get the lock, but there might be a thread that is continuously failing to get the lock. Lamport's bakery lock is starvation-free. It ensures that threads can acquire locks in the order of their requests.

However, as noted by Herlihy and Shavit [11], the above definitions based on locks are unsatisfactory, because it is often difficult to identify a particular field in the object as a lock. Instead, they suggest defining them in terms of method calls. They also notice that the above definitions implicitly assume that every thread acquiring the lock will eventually release it. This assumption requires *fair* scheduling, *i.e.*, every thread gets eventually executed.

Following Herlihy and Shavit [11], we say an object is *deadlock-free*, if in each *fair* execution there always exists some method call that can finish. As an example in Figure 2(d), we use a test-and-set lock to synchronize the increments of the counter. Since some thread is guaranteed to acquire the test-and-set lock, the method call of that thread is guaranteed to finish. Thus the object is deadlock-free. Similarly, a *starvation-free* object guarantees that every method call can finish in fair executions. Figure 2(e) shows a counter implemented with Lamport's bakery lock. It is starvation-free since the bakery lock ensures that

	Wait-Free	Lock-Free	Obstruction-Free	Deadlock-Free	Starvation-Free
Π_A	(t, Div.)	Div.	Div.	Div.	(t, Div.)
Π	(t, Div.)	Div.	Div. if Isolating	Div. if Fair	(t, Div.) if Fair

Table 1: Characterizing Progress Properties via Contextual Refinements $\Pi \sqsubseteq \Pi_A$

every thread can acquire the lock and hence every method call can eventually complete.

Our Results. None of the above definitions of the five progress properties describes their guarantees regarding the behaviors of client code. In this paper, we define several contextual refinements to characterize the effects over client behaviors when the client uses objects with some progress properties. We show that linearizability together with a progress property is equivalent to a certain termination-sensitive contextual refinement. Table 1 summarizes our results.

For each progress property, the new contextual refinement $\Pi \sqsubseteq \Pi_A$ is defined with respect to a divergence behavior and/or a specific scheduling at the implementation level (the third row in Table 1) and at the abstract side (the second row), in addition to the I/O events in the basic contextual refinement for linearizability.

- For wait-freedom, we need to observe the divergence of each individual thread t, represented by "(t, Div.)" in Table 1, at both the concrete and the abstract levels. We show that, if the thread t of a client program diverges when the client uses a linearizable and wait-free object Π, then thread t must also diverge when using Π_A instead.
- The case for lock-freedom is similar, except that we now consider the divergence behaviors of the *whole* client program rather than individual threads (denoted by "Div." in Table 1). If a client diverges when using a linearizable and lock-free object Π, it must also diverge when it uses Π_A instead.
- For obstruction-freedom, we consider the behaviors of *isolating* executions at the concrete side (denoted by "Div. if Isolating" in Table 1). In those executions, eventually only one thread is running. We show that, if a client diverges in an isolating execution when it uses a linearizable and obstruction-free object Π, it must also diverge in some abstract execution.
- For deadlock-freedom, we only care about *fair* executions at the concrete level (denoted by "Div. if Fair" in Table 1).
- For starvation-freedom, we observe the divergence of each individual thread at both levels and restrict our considerations to fair executions for the concrete side ("(t, Div.) if Fair" in Table 1). Any thread using Π can diverge in a fair execution, only if it also diverges in some abstract execution.

These new contextual refinements can characterize linearizable objects with progress properties. We will formalize the results and give examples in Section 5.

$$
\begin{array}{rrcl}
(\mathit{Expr}) & E & ::= & x \mid n \mid E + E \mid \ldots \\
(\mathit{BExp}) & B & ::= & \mathbf{true} \mid \mathbf{false} \mid E = E \mid !B \mid \ldots \\
(\mathit{Instr}) & c & ::= & x := E \mid x := [E] \mid [E] := E \mid \mathbf{print}(E) \\
 & & \mid & x := \mathbf{cons}(E, \ldots, E) \mid \mathbf{dispose}(E) \mid \ldots \\
(\mathit{Stmt}) & C & ::= & \mathbf{skip} \mid c \mid x := f(E) \mid \mathbf{return}\ E \mid \mathbf{fret}(n) \mid \mathbf{noret} \\
 & & \mid & \mathbf{end} \mid \langle C \rangle \mid C; C \mid \mathbf{if}\ (B)\ C\ \mathbf{else}\ C \mid \mathbf{while}\ (B)\{C\} \\
(\mathit{Prog}) & W & ::= & \mathbf{skip} \mid \mathbf{let}\ \Pi\ \mathbf{in}\ C \| \ldots \| C \\
(\mathit{ODecl}) & \Pi & ::= & \{f_1 \rightsquigarrow (x_1, C_1), \ldots, f_n \rightsquigarrow (x_n, C_n)\}
\end{array}
$$

Fig. 3: Syntax of the Programming Language

3 Formal Setting and Linearizability

In this section, we formalize linearizability and show its equivalence to a contextual refinement that preserves safety properties. This equivalence is the basis of our new results that relate contextual refinement and progress properties.

Language and Semantics We use a similar language as in previous work of Liang and Feng [13]. As shown in Figure 3, a program W consists of several client threads that run in parallel. Each thread could call the methods declared in the object Π. A method f is defined as a pair (x, C), where x is the formal argument and C is the method body. We write $f \rightsquigarrow (x, C)$. The object Π could be either concrete with fine-grained code that we want to verify, or abstract (usually denoted as Π_A in the following) that we consider as the specification. For the latter case, each method body should be an atomic operation of the form $\langle C \rangle$ and it should be always safe to execute it. For simplicity, we assume there is only one object in the program W and each method takes one argument only. However, it is easy to extend our work to multiple objects and arguments.

We use the command **noret** at the end of methods that terminate but do not execute **return** E. It is automatically appended to the method code and is not supposed to be used by programmers. The command **return** E will first calculate the return value n and reduce to $\mathbf{fret}(n)$, another runtime command automatically generated during executions. We separate the evaluation of E from returning its value n to the client, to allow interference between the two steps. Note that the atomic block $\langle C \rangle$ may contain the command **return** E. In that case, $\langle C \rangle$ would also reduce to $\mathbf{fret}(n)$.

To discuss progress properties later, we introduce an auxiliary command **end**. It is a special marker that can be added at the end of a thread, but should not be used directly by programmers. Other commands are mostly standard. Clients can use $\mathbf{print}(E)$ to produce observable external events. We do not allow the object's methods to produce external events. To simplify the semantics, we also assume there are no nested method calls.

Figure 4 defines program states and event traces. We partition a global state $\mathcal{S}$ into the client memory σ_c, the object σ_o, and a thread pool $\mathcal{K}$. A client can only access the client memory σ_c, unless it calls object methods. The thread pool maps each thread ID $\mathtt{t}$ to its local call stack frame. A call stack κ could be either

$$
\begin{array}{rrcl}
(\mathit{ThrdID}) & \mathtt{t} & \in & \mathit{Nat} \\
(\mathit{Mem}) & \sigma & \in & (\mathit{PVar} \cup \mathit{Nat}) \rightharpoonup \mathit{Int} \\
(\mathit{CallStk}) & \kappa & ::= & (\sigma_l, x, C) \mid \circ \\
(\mathit{ThrdPool}) & \mathcal{K} & ::= & \{\mathtt{t}_1 \rightsquigarrow \kappa_1, \ldots, \mathtt{t}_n \rightsquigarrow \kappa_n\} \\
(\mathit{PState}) & \mathcal{S} & ::= & (\sigma_c, \sigma_o, \mathcal{K}) \\
(\mathit{LState}) & s & ::= & (\sigma_c, \sigma_o, \kappa)
\end{array}
$$

$$
\begin{array}{rrcl}
(\mathit{Evt}) & e & ::= & (\mathtt{t}, f, n) \mid (\mathtt{t}, \mathbf{ret}, n) \\
 & & \mid & (\mathtt{t}, \mathbf{obj}) \mid (\mathtt{t}, \mathbf{obj}, \mathbf{abort}) \\
 & & \mid & (\mathtt{t}, \mathbf{out}, n) \mid (\mathtt{t}, \mathbf{clt}) \\
 & & \mid & (\mathtt{t}, \mathbf{clt}, \mathbf{abort}) \mid (\mathtt{t}, \mathbf{term}) \\
 & & \mid & (\mathbf{spawn}, n) \\
(\mathit{ETrace}) & T & ::= & \epsilon \mid e :: T \quad \text{(co-inductive)}
\end{array}
$$

Fig. 4: States and Event Traces

empty ($\circ$) when the thread is not executing a method, or a triple (σ_l, x, C), where σ_l maps the method's formal argument and local variables to their values, x is the caller's variable to receive the return value, and C is the caller's remaining code to be executed after the method returns. To give a thread-local semantics, we also define the thread local view s of the state that only includes one call stack.

Figure 5 contains selected rules of the operational semantics. To describe the operational semantics for threads, we use an execution context $\mathbf{E}$, where $\mathbf{E} ::= [\,] \mid \mathbf{E}; C$. The execution of code occurs in the hole $[\,]$. The context $\mathbf{E}[\,C\,]$ results from placing C into the hole.

We have three kinds of transitions. We write $(W, \mathcal{S}) \overset{e}{\longmapsto} (W', \mathcal{S}')$ for the top-level program transitions and $(C, s) \overset{e}{\longrightarrow}_{\mathtt{t},\Pi} (C', s')$ for the transitions of thread $\mathtt{t}$ with the object Π. We also introduce the local transition $(C, \sigma) \longrightarrow\!\!\triangleright_{\mathtt{t}} (C', \sigma')$ to describe a step inside or outside method calls of concurrent objects. It accesses only object memory and method local variables (for the case inside method calls), or only client memory (for the other case). We then lift a local transition to a thread transition that produces an event $(\mathtt{t}, \mathbf{obj})$ or $(\mathtt{t}, \mathbf{clt})$. All three transitions also include steps that lead to the error state **abort**.

We define all the generated events e in Figure 4. A method invocation event $(\mathtt{t}, f, n)$ is produced when thread $\mathtt{t}$ executes $x := f(E)$, where the argument E's value is n. A return $(\mathtt{t}, \mathbf{ret}, n)$ is produced with the return value n. $\mathbf{print}(E)$ generates an output $(\mathtt{t}, \mathbf{out}, n)$, and **end** generates a termination marker $(\mathtt{t}, \mathbf{term})$. Other steps generate either normal object actions $(\mathtt{t}, \mathbf{obj})$ (for steps inside method calls) or silent client actions $(\mathtt{t}, \mathbf{clt})$ (for client steps other than $\mathbf{print}(E)$). For transitions leading to the error state **abort**, fault events are produced: $(\mathtt{t}, \mathbf{obj}, \mathbf{abort})$ by the object method code and $(\mathtt{t}, \mathbf{clt}, \mathbf{abort})$ by the client code. We also introduce an auxiliary event $(\mathbf{spawn}, n)$ to represent spawning n threads. It is automatically inserted at the beginning of a generated event trace, according to the total number of threads in the program.[3] Note that in this paper, we follow Herlihy and Wing [12] and model dynamic thread creation by simply treating each child thread as an additional thread that executes no operations before being created. Outputs and faults are observable events. We write $\mathsf{tid}(e)$ for the thread ID in the event e. The predicate $\mathsf{is\_clt}(e)$ states

[3] The spawning event $(\mathbf{spawn}, n)$ is newly introduced in this TR. It helps to hide the parameter of the total number of threads in the fairness definition in the submitted version, and to formulate the alternative definitions of progress properties.

$$\frac{(C_i, (\sigma_c, \sigma_o, \mathcal{K}(i))) \xrightarrow{e}_{i,\Pi} (C_i', (\sigma_c', \sigma_o', \kappa'))}{(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots C_i \ldots \|\, C_n, (\sigma_c, \sigma_o, \mathcal{K})) \overset{e}{\longmapsto} (\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots C_i' \ldots \|\, C_n, (\sigma_c', \sigma_o', \mathcal{K}\{i \rightsquigarrow \kappa'\}))}$$

(a) Program Transitions

$$\frac{\Pi(f) = (y, C) \qquad [\![E]\!]_{\sigma_c} = n \qquad x \in dom(\sigma_c) \qquad \kappa = (\{y \rightsquigarrow n\}, x, \mathbf{E}[\,\mathbf{skip}\,])}{(\mathbf{E}[\,x := f(E)\,], (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathrm{t}, f, n)}_{\mathrm{t},\Pi} (C; \mathbf{noret}, (\sigma_c, \sigma_o, \kappa))}$$

$$\frac{f \notin dom(\Pi) \quad \text{or} \quad [\![E]\!]_{\sigma_c}\ \textit{undefined} \quad \text{or} \quad x \notin dom(\sigma_c)}{(\mathbf{E}[\,x := f(E)\,], (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathrm{t},\mathbf{clt},\mathbf{abort})}_{\mathrm{t},\Pi} \mathbf{abort}}$$

$$\frac{\kappa = (\sigma_l, x, C) \qquad \sigma_c' = \sigma_c\{x \rightsquigarrow n\}}{(\mathbf{fret}(n), (\sigma_c, \sigma_o, \kappa)) \xrightarrow{(\mathrm{t},\mathbf{ret},n)}_{\mathrm{t},\Pi} (C, (\sigma_c', \sigma_o, \circ))} \qquad \frac{}{(\mathbf{end}, s) \xrightarrow{(\mathrm{t},\mathbf{term})}_{\mathrm{t},\Pi} (\mathbf{skip}, s)}$$

$$\frac{[\![E]\!]_{\sigma_c} = n}{(\mathbf{E}[\,\mathbf{print}(E)\,], (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathrm{t},\mathbf{out},n)}_{\mathrm{t},\Pi} (\mathbf{E}[\,\mathbf{skip}\,], (\sigma_c, \sigma_o, \circ))}$$

$$\frac{(C, \sigma_o \uplus \sigma_l) \longrightarrow_{\mathrm{t}} (C', \sigma_o' \uplus \sigma_l') \qquad dom(\sigma_l) = dom(\sigma_l')}{(C, (\sigma_c, \sigma_o, (\sigma_l, x, C_c))) \xrightarrow{(\mathrm{t},\mathbf{obj})}_{\mathrm{t},\Pi} (C', (\sigma_c, \sigma_o', (\sigma_l', x, C_c)))}$$

$$\frac{(C, \sigma_o \uplus \sigma_l) \longrightarrow_{\mathrm{t}} \mathbf{abort}}{(C, (\sigma_c, \sigma_o, (\sigma_l, x, C_c))) \xrightarrow{(\mathrm{t},\mathbf{obj},\mathbf{abort})}_{\mathrm{t},\Pi} \mathbf{abort}} \qquad \frac{(C, \sigma_c) \longrightarrow_{\mathrm{t}} (C', \sigma_c')}{(C, (\sigma_c, \sigma_o, \circ)) \xrightarrow{(\mathrm{t},\mathbf{clt})}_{\mathrm{t},\Pi} (C', (\sigma_c', \sigma_o, \circ))}$$

(b) Thread Transitions

$$\frac{[\![E]\!]_{\sigma} = n}{(\mathbf{E}[\,\mathbf{return}\ E\,], \sigma) \longrightarrow_{\mathrm{t}} (\mathbf{fret}(n), \sigma)} \qquad \frac{}{(\mathbf{noret}, \sigma) \longrightarrow_{\mathrm{t}} \mathbf{abort}}$$

$$\frac{(C, \sigma) \longrightarrow_{\mathrm{t}}^{*} (\mathbf{skip}, \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathrm{t}} (\mathbf{E}[\,\mathbf{skip}\,], \sigma')} \qquad \frac{(C, \sigma) \longrightarrow_{\mathrm{t}}^{*} (\mathbf{fret}(n), \sigma')}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathrm{t}} (\mathbf{fret}(n), \sigma')} \qquad \frac{(C, \sigma) \longrightarrow_{\mathrm{t}}^{*} \mathbf{abort}}{(\mathbf{E}[\,\langle C \rangle\,], \sigma) \longrightarrow_{\mathrm{t}} \mathbf{abort}}$$

(c) Local Thread Transitions

Fig. 5: Selected Rules of Operational Semantics

that the event e is either a silent client action, an output, or a client fault. We write $\mathsf{is\_inv}(e)$ and $\mathsf{is\_ret}(e)$ to denote that the event e is a method invocation and a return, respectively. The predicate $\mathsf{is\_res}(e)$ denotes a return or an object fault, and $\mathsf{is\_abt}(e)$ denotes a fault of the object or the client. Other predicates are similar and summarized below.

- $\mathsf{is\_inv}(e)$ iff there exist t, f and n such that $e = (\mathsf{t}, f, n)$;
- $\mathsf{is\_ret}(e)$ iff there exist t and n' such that $e = (\mathsf{t}, \mathbf{ret}, n')$;
- $\mathsf{is\_obj\_abt}(e)$ iff there exists t such that $e = (\mathsf{t}, \mathbf{obj}, \mathbf{abort})$;
- $\mathsf{is\_res}(e)$ iff either $\mathsf{is\_ret}(e)$ or $\mathsf{is\_obj\_abt}(e)$ holds;
- $\mathsf{is\_obj}(e)$ iff either $e = (\_, \mathbf{obj})$ or $\mathsf{is\_inv}(e)$ or $\mathsf{is\_res}(e)$ holds;
- $\mathsf{is\_clt\_abt}(e)$ iff there exists t such that $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$;
- $\mathsf{is\_abt}(e)$ iff either $\mathsf{is\_obj\_abt}(e)$ or $\mathsf{is\_clt\_abt}(e)$ holds;
- $\mathsf{is\_clt}(e)$ iff there exists t and n such that either $e = (\mathsf{t}, \mathbf{clt})$ or $e = (\mathsf{t}, \mathbf{out}, n)$ or $e = (\mathsf{t}, \mathbf{clt}, \mathbf{abort})$ holds.

An event trace T is a finite or infinite sequence of events. We write $T(i)$ for the i-th event of T. $\mathsf{last}(T)$ is the last event in a finite T. The trace $T(1..i)$ is the sub-trace $T(1), \ldots, T(i)$ of T, and $|T|$ is the length of T ($|T| = \omega$ if T is infinite). The trace $T|_{\mathsf{t}}$ represents the sub-trace of T consisting of all events whose thread ID is t. We generate event traces from executions in Figure 6. We write $\mathcal{T}[\![W, (\sigma_c, \sigma_o)]\!]$ for the prefix-closed set of finite traces produced by the executions of W with the initial client memory σ_c, the object σ_o, and empty call stacks for all threads. Similarly, we write $\mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o)]\!]$ for the finite or infinite event traces produced by complete executions. In the definitions, we use the notation $\_ \overset{T}{\longmapsto}{}^* \_$ for zero or multiple-step program transitions the generate the trace T. Similarly, $\_ \overset{T}{\longmapsto}{}^\omega \cdot$ denotes the existence of an infinite T-labelled execution. Note that by using $\lfloor W \rfloor$, **end** is automatically appended at the end of each thread in W to explicitly mark the termination of a thread. Using $\lfloor T \rfloor_W$, we insert the spawning event $(\mathbf{spawn}, n)$ at the beginning of T, where n is the total number of threads in W. Then we could use $\mathsf{tnum}(T)$ to get the number of threads in the program that generates T. Figure 6 also shows various ways to get histories and observable behaviors of a program, which we will explain later.

Linearizability and Basic Contextual Refinement Linearizability [12] is defined using histories. Histories are special event traces only consisting of method invocation, method return, and object faults.

We say a response e_2 *matches* an invocation e_1, denoted as $\mathsf{match}(e_1, e_2)$, iff they have the same thread ID.

$$\mathsf{match}(e_1, e_2) \overset{\text{def}}{=} \mathsf{is\_inv}(e_1) \wedge \mathsf{is\_res}(e_2) \wedge (\mathsf{tid}(e_1) = \mathsf{tid}(e_2))$$

A history T is *sequential*, *i.e.*, $\mathsf{seq}(T)$, iff the first event of T is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. It is inductively defined as follows.

$$\frac{}{\mathsf{seq}(\epsilon)} \qquad \frac{\mathsf{is\_inv}(e)}{\mathsf{seq}(e :: \epsilon)} \qquad \frac{\mathsf{match}(e_1, e_2) \qquad \mathsf{seq}(T)}{\mathsf{seq}(e_1 :: e_2 :: T)}$$

$$\mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\lfloor T\rfloor_W \mid \exists W',\mathcal{S}'.\ (\lfloor W\rfloor,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{*} (W',\mathcal{S}') \vee (\lfloor W\rfloor,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{*} \mathbf{abort}\}$$

$$\mathcal{T}_\omega[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\lfloor T\rfloor_W \mid (\lfloor W\rfloor,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{\omega}\cdot \vee (\lfloor W\rfloor,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{*} (\mathbf{skip},\_) \vee (\lfloor W\rfloor,(\sigma_c,\sigma_o,\circledcirc)) \xmapsto{T}{}^{*} \mathbf{abort}\}$$

$$\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n \rfloor \stackrel{\text{def}}{=} \mathbf{let}\ \Pi\ \mathbf{in}\ (C_1;\mathbf{end}) \| \ldots \| (C_n;\mathbf{end})$$

$$\lfloor T\rfloor_{(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1\|\ldots\|C_n)} \stackrel{\text{def}}{=} (\mathbf{spawn},n)::T \qquad \mathsf{tnum}((\mathbf{spawn},n)::T) \stackrel{\text{def}}{=} n$$

$$\circledcirc \stackrel{\text{def}}{=} \{\mathsf{t}_1 \rightsquigarrow \circ,\ldots,\mathsf{t}_n \rightsquigarrow \circ\} \qquad \mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (\,|(T|_{\mathsf{t}})| = \omega\,)\}$$

$$\mathsf{iso}(T) \ \text{ iff } \ |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})$$

$$\mathsf{fair}(T) \ \text{ iff } \ |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t},\mathbf{term})$$

$$\mathcal{H}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_hist}(T) \mid T \in \mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]\}$$

$$\mathcal{O}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}[\![W,(\sigma_c,\sigma_o)]\!]\}$$

$$\mathcal{O}_{t\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T),\mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_\omega[\![W,(\sigma_c,\sigma_o)]\!]\}$$

$$\mathcal{O}_{\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_\omega[\![W,(\sigma_c,\sigma_o)]\!]\}$$

$$\mathcal{O}_{i\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_\omega[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{iso}(T)\}$$

$$\mathcal{O}_{f\omega}[\![W,(\sigma_c,\sigma_o)]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid \exists n.\ T \in \mathcal{T}_\omega[\![W,(\sigma_c,\sigma_o)]\!] \wedge \mathsf{fair}(T)\}$$

Fig. 6: Generation of Event Traces

Then T is *well-formed* iff, for all t, $T|_{\mathsf{t}}$ is sequential.

$$\mathsf{well\_formed}(T) \stackrel{\text{def}}{=} \forall \mathsf{t}.\ \mathsf{seq}(T|_{\mathsf{t}})\,.$$

T is *complete* iff it is well-formed and every invocation has a matching response. An invocation is *pending* if no matching response follows it. We write $\mathsf{pend\_inv}(T)$ for the set of pending invocations in T.

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e{=}T(i) \wedge \mathsf{is\_inv}(e) \wedge (\forall j.\ i{<}j{\leq}|T| \Rightarrow \neg\mathsf{match}(e,T(j)))\}$$

We handle pending invocations in an incomplete history T following the standard linearizability definition [12]: We append zero or more return events to T, and drop the remaining pending invocations. Then we get a set of complete histories, which is denoted by $\mathsf{completions}(T)$. Formally, we define $\mathsf{completions}(T)$ as follows.

Definition 1 (Extensions of a history). $\mathsf{extensions}(T)$ *is a set of well-formed histories where we extend* T *by appending successful return events:*

$$\frac{\mathsf{well\_formed}(T)}{T \in \mathsf{extensions}(T)} \qquad \frac{T' \in \mathsf{extensions}(T) \qquad \mathsf{is\_ret}(e) \qquad \mathsf{well\_formed}(T'::e)}{T'::e \in \mathsf{extensions}(T)}$$

Or equivalently,

$$\mathsf{extensions}(T) \stackrel{\text{def}}{=} \{T' \mid \mathsf{well\_formed}(T') \wedge \exists T_{ok}.\ T'{=}T::T_{ok} \wedge \forall i.\ \mathsf{is\_ret}(T_{ok}(i))\}.$$

Definition 2 (Completions of a history). $\mathsf{truncate}(T)$ *is the maximal complete sub-history of* T*, which is inductively defined by dropping the pending invocations in* T*:*

$$\mathsf{truncate}(\epsilon) \stackrel{\text{def}}{=} \epsilon$$

$$\mathsf{truncate}(e :: T) \stackrel{\text{def}}{=} \begin{cases} e :: \mathsf{truncate}(T) & \textit{if } \mathsf{is\_res}(e) \textit{ or } \exists i.\ \mathsf{match}(e, T(i)) \\ \mathsf{truncate}(T) & \textit{otherwise} \end{cases}$$

Then $\mathsf{completions}(T) \stackrel{\text{def}}{=} \{\mathsf{truncate}(T') \mid T' \in \mathsf{extensions}(T)\}$*. It's a set of histories without pending invocations.*

Then we can formulate the linearizability relation between well-formed histories, which is a core notion used in the linearizability definition of an object.

Definition 3 (Linearizable Histories). $T \preceq_{\mathsf{lin}} T'$ *iff*

1. $\forall \mathsf{t}.\ T|_{\mathsf{t}} = T'|_{\mathsf{t}}$*;*
2. *there exists a bijection* $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T'|\}$ *such that* $\forall i.\ T(i) = T'(\pi(i))$ *and* $\forall i, j.\ i < j \wedge \mathsf{is\_ret}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j)$*.*

That is, T is linearizable *w.r.t.* T' if the latter is a permutation of the former, preserving the order of events in the same threads and the order of the non-overlapping method calls. Then an *object* is linearizable iff all its concurrent histories after completions are linearizable *w.r.t.* some *legal sequential* histories. We use $\Pi_A \rhd (\sigma_a, T')$ to mean that T' is a legal sequential history generated by any client using the specification Π_A with an initial abstract object σ_a.

$$\Pi_A \rhd (\sigma_a, T) \stackrel{\text{def}}{=}$$
$$\exists n, C_1, \ldots, C_n, \sigma_c.\ T \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a)]\!] \wedge \mathsf{seq}(T)$$

As defined in Figure 6, we use $\mathcal{H}[\![W, (\sigma_c, \sigma_a)]\!]$ to generate histories from W, where $\mathsf{get\_hist}(T)$ projects the event trace T to the sub-history.

Definition 4 (Linearizability of Objects). *The object's implementation* Π *is linearizable w.r.t.* Π_A *under a refinement mapping* φ*, denoted by* $\Pi \preceq_{\varphi} \Pi_A$ *iff*
$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a, T.\ T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge (\varphi(\sigma_o) = \sigma_a)$
$\implies \exists T_c, T'.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\sigma_a, T') \wedge T_c \preceq_{\mathsf{lin}} T'$

Here the mapping φ relates concrete objects to abstract ones:

$$(\mathit{RefMap})\quad \varphi \in \mathit{Mem} \rightharpoonup \mathit{AbsObj}$$

The side condition $\varphi(\sigma_o) = \theta$ in the above definition requires the initial concrete object σ_o to be a well-formed data structure representing a valid object θ.

Next we define a contextual refinement between the concrete object and its specification, which is equivalent to linearizability. Informally, this contextual refinement states that for any set of client threads, the program W has no more observable behaviors than the corresponding abstract program. Below we use $\mathcal{O}[\![W, (\sigma_c, \sigma_o)]\!]$ to represent the set of observable event traces generated during the executions of W with the initial state (σ_c, σ_o) (and empty stacks). It is defined similarly as $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$ in Figure 6, but now the traces consist of observable events only (outputs, client faults or object faults).

Definition. An object Π satisfies P under a refinement mapping φ, $P_\varphi(\Pi)$, iff
$\forall n, C_1, \ldots, C_n, \mathcal{S}, T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n), \mathcal{S}]\!] \wedge (\mathcal{S} \in dom(\varphi)) \implies P(T)$.

$$\mathcal{T}_\omega[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathbf{spawn}, |W|) :: T \mid (\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{\omega} \cdot \vee (\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{*} (\mathbf{skip}, \_) \vee (\lfloor W \rfloor, \mathcal{S}) \xrightarrow{T}{}^{*} \mathbf{abort}\}$$

$$\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n \rfloor \stackrel{\text{def}}{=} \mathbf{let}\ \Pi\ \mathbf{in}\ (C_1; \mathbf{end}) \| \ldots \| (C_n; \mathbf{end})$$

$$|\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n| \stackrel{\text{def}}{=} n \qquad \mathsf{tnum}((\mathbf{spawn}, n) :: T) \stackrel{\text{def}}{=} n$$

$$\mathsf{pend\_inv}(T) \stackrel{\text{def}}{=} \{e \mid \exists i.\ e = T(i) \wedge \mathsf{is\_inv}(e) \wedge \neg \exists j.\ (j > i \wedge \mathsf{match}(e, T(j)))\}$$

$$\mathsf{prog\text{-}t}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{match}(e, T(j))$$

$$\mathsf{prog\text{-}s}(T) \ \text{ iff } \ \forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \implies \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))$$

$$\mathsf{abt}(T) \ \text{ iff } \ \exists i.\ \mathsf{is\_abt}(T(i))$$

$$\mathsf{sched}(T) \ \text{ iff } \ |T| = \omega \wedge \mathsf{pend\_inv}(T) \neq \emptyset \implies \exists e.\ e \in \mathsf{pend\_inv}(T) \wedge |(T|_{\mathsf{tid}(e)})| = \omega$$

$$\mathsf{fair}(T) \ \text{ iff } \ |T| = \omega \implies \forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| = \omega \vee \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$$

$$\mathsf{iso}(T) \ \text{ iff } \ |T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})$$

$$\mathsf{wait\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt} \qquad \mathsf{starvation\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

$$\mathsf{lock\text{-}free} \ \text{ iff } \ \mathsf{sched} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt} \qquad \mathsf{deadlock\text{-}free} \ \text{ iff } \ \mathsf{fair} \implies \mathsf{prog\text{-}s} \vee \mathsf{abt}$$

$$\mathsf{obstruction\text{-}free} \ \text{ iff } \ \mathsf{sched} \wedge \mathsf{iso} \implies \mathsf{prog\text{-}t} \vee \mathsf{abt}$$

Fig. 7: Formalizing Progress Properties

Definition 5 (Basic Contextual Refinement). *$\Pi \sqsubseteq_\varphi \Pi_A$ iff*
$\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a)$
$\implies \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \| \ldots \| C_n), (\sigma_c, \sigma_o)]\!] \subseteq \mathcal{O}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \| \ldots \| C_n), (\sigma_c, \sigma_a)]\!]$.

Following Filipović *et al.* [4], we can prove that linearizability is equivalent to this contextual refinement. We give the proofs in Appendix B.1.

Theorem 1 (Basic Equivalence). *$\Pi \preceq_\varphi \Pi_A \iff \Pi \sqsubseteq_\varphi \Pi_A$.*

Theorem 1 allows us to use $\Pi \sqsubseteq_\varphi \Pi_A$ to identify linearizable objects. However, we cannot use it to observe progress properties of objects. For the following example, $\Pi \sqsubseteq_\varphi \Pi_A$ holds although no concrete method call of `f` could finish (we assume this object contains a method `f` only).

Π(`f`) : `while(true) skip;` $\quad$ Π_A(`f`) : `skip;` $\quad$ C : `print(1); f(); print(1);`

The reason is that $\Pi \sqsubseteq_\varphi \Pi_A$ considers a *prefix-closed* set of event traces at the abstract side. For the above client C, the observable behaviors of **let** Π **in** C can all be found in the prefix-closed set of behaviors produced by **let** Π_A **in** C.

4 Formalizing Progress Properties

We define progress in Figure 7 as properties over both event traces T and object implementations Π. We say an object implementation Π has a progress property

$$\mathsf{lock\text{-}free} \iff \mathsf{wait\text{-}free} \vee \mathsf{prog\text{-}s} \qquad \mathsf{starvation\text{-}free} \iff \mathsf{wait\text{-}free} \vee \neg\mathsf{fair}$$
$$\mathsf{obstruction\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{iso} \qquad \mathsf{deadlock\text{-}free} \iff \mathsf{lock\text{-}free} \vee \neg\mathsf{fair}$$

Fig. 8: Relationships between Progress Properties

P iff all its event traces have the property. Here we use $\mathcal{T}_\omega$ to generate the event traces. Its definition in Figure 7 is similar to $\mathcal{T}[\![W, \mathcal{S}]\!]$ of Figure **??**, but $\mathcal{T}_\omega[\![W, \mathcal{S}]\!]$ is for the set of finite or infinite event traces produced by *complete* executions. We use $(W, \mathcal{S}) \xrightarrow{T}{}^{\omega} \cdot$ to denote the existence of a T-labelled infinite execution. $(W, \mathcal{S}) \xrightarrow{T}{}^{*} (\mathbf{skip}, \_)$ represents a terminating execution that produces T. By using $\lfloor W \rfloor$, we append **end** at the end of each thread to explicitly mark the termination of the thread. We also insert the spawning event $(\mathbf{spawn}, n)$ at the beginning of T, where n is the number of threads in W. Then we can use $\mathsf{tnum}(T)$ to get the number n, which is needed to define fairness, as shown below.

Before formulating each progress property over event traces, we first define some auxiliary properties in Figure 7. $\mathsf{prog\text{-}t}(T)$ guarantees that every method call in T eventually finishes. $\mathsf{prog\text{-}s}(T)$ guarantees that *some* pending method call finishes. Different from prog-t, the return event $T(j)$ in prog-s does not have to be a matching return of the pending invocation e. $\mathsf{abt}(T)$ says that T ends with a fault event.

There are three useful conditions on scheduling. The basic requirement for a good schedule is sched. If T is infinite and there exist pending calls, then at least one pending thread should be scheduled infinitely often. In fact, there are two possible reasons causing a method call of thread t to pend. Either t is no longer scheduled, or it is always scheduled but the method call never finishes. sched rules out the bad schedule where no thread with an invoked method is active. For instance, the following infinite trace does *not* satisfy sched.

$$(\mathsf{t}_1, f_1, n_1) :: (\mathsf{t}_2, f_2, n_2) :: (\mathsf{t}_1, \mathbf{obj}) :: (\mathsf{t}_3, \mathbf{clt}) :: (\mathsf{t}_3, \mathbf{clt}) :: \ldots$$

If T is infinite, $\mathsf{fair}(T)$ requires every non-terminating thread be scheduled infinitely often; and $\mathsf{iso}(T)$ requires eventually only one thread be scheduled. We can see that a fair schedule is a good schedule satisfying sched.

At the bottom of Figure 7 we define the progress properties formally. We omit the parameter T in the formulae to simplify the presentation. An event trace T is wait-free (*i.e.*, $\mathsf{wait\text{-}free}(T)$ holds) if under the good schedule sched, it guarantees prog-t unless it ends with a fault. $\mathsf{lock\text{-}free}(T)$ is similar except that it guarantees prog-s. Starvation-freedom and deadlock-freedom guarantee prog-t and prog-s under fair scheduling. Obstruction-freedom guarantees prog-t if some pending thread is always scheduled (sched) and runs in isolation (iso).

Figure 8 contains lemmas that relate progress properties. For instance, an event trace is starvation-free, iff it is wait-free or not fair. These lemmas give us the relationship lattice in Figure 1. To close the lattice, we also define a progress property in the sequential setting. *Sequential termination* guarantees that every method call must finish in a trace produced by a sequential client. The formal

$$\mathsf{div\_tids}(T) \stackrel{\text{def}}{=} \{\mathsf{t} \mid (|(T|_{\mathsf{t}})| = \omega)\}$$
$$\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{i\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{iso}(T)\}$$
$$\mathcal{O}_{f\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$
$$\mathcal{O}_{t\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]\}$$
$$\mathcal{O}_{ft\omega}[\![W,\mathcal{S}]\!] \stackrel{\text{def}}{=} \{(\mathsf{get\_obsv}(T), \mathsf{div\_tids}(T)) \mid T \in \mathcal{T}_{\omega}[\![W,\mathcal{S}]\!] \wedge \mathsf{fair}(T)\}$$

Fig. 9: Generation of Complete Event Traces

definition is given in the companion TR [15], and we prove that it is implied by each of the five progress properties for concurrent objects.

5 Equivalence to Contextual Refinements

We extend the basic contextual refinement in Definition 5 to observe progress as well as linearizability. For each progress property, we carefully choose the observable behaviors at the concrete and the abstract levels.

5.1 Observable Behaviors

In Figure 9, we define various observable behaviors for the termination-sensitive contextual refinements.

We use $\mathcal{O}_{\omega}[\![W,\mathcal{S}]\!]$ to represent the set of observable event traces produced by complete executions of $(W,\mathcal{S})$. Recall that $\mathsf{get\_obsv}(T)$ gets the sub-trace of T consisting of all the observable events only. Unlike the prefix-closed set $\mathcal{O}[\![W,\mathcal{S}]\!]$, this definition utilizes $\mathcal{T}_{\omega}[\![W,\mathcal{S}]\!]$ (see Figure 7) whose event traces are all complete and could be infinite. Thus it allows us to observe divergence of the whole program. $\mathcal{O}_{i\omega}$ and $\mathcal{O}_{f\omega}$ take the complete observable traces of *isolating* and fair executions respectively. Here $\mathsf{iso}(T)$ and $\mathsf{fair}(T)$ are defined in Figure 7.

We could also observe divergence of individual threads rather than the whole program. We define $\mathsf{div\_tids}(T)$ to collect the set of threads that diverge in the trace T. Then we write $\mathcal{O}_{t\omega}[\![W,\mathcal{S}]\!]$ to get both the observable behaviors and the diverging threads in the complete executions. $\mathcal{O}_{ft\omega}[\![W,\mathcal{S}]\!]$ is defined similarly but considers fair executions only.

More on divergence. In general, divergence means non-termination. For example, we could say that the following two-threaded program (5.1) must diverge since it never terminates.

```
x := x + 1;    ||    while(true) skip;        (5.1)
```

But for individual threads, divergence is not equivalent to non-termination, since a non-terminating thread may either have an infinite execution or simply be not scheduled from some point due to unfair scheduling. We view only the former case as divergence. For instance, in an unfair execution, the left thread of (5.1)

P	wait-free	lock-free	obstruction-free	deadlock-free	starvation-free
$\Pi \sqsubseteq_{\varphi}^{P} \Pi_A$	$\mathcal{O}_{t\omega} \subseteq \mathcal{O}_{t\omega}$	$\mathcal{O}_{\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{i\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{f\omega} \subseteq \mathcal{O}_{\omega}$	$\mathcal{O}_{ft\omega} \subseteq \mathcal{O}_{t\omega}$

Table 2: Contextual Refinements $\Pi \sqsubseteq_{\varphi}^{P} \Pi_A$ for Progress Properties P

may never be scheduled and hence it has no chance to terminate. It does not diverge. Similarly, for the following program (5.2),

```
while(true) skip;    ||    while(true) skip;
```
(5.2)

the whole program must diverge, but it is possible that a single thread does not diverge in an execution.

5.2 New Contextual Refinements and Equivalence Results

In Table 2, we summarize the definitions of the termination-sensitive contextual refinements. Each new contextual refinement follows the basic one in Definition 5 but takes different observable behaviors as specified in Table 2. For example, the contextual refinement for wait-freedom is formally defined as follows:

$$\Pi \sqsubseteq_{\varphi}^{\mathsf{wait\text{-}free}} \Pi_A \;\text{ iff }\; (\forall n, C_1, \ldots, C_n, \mathcal{S}, \mathcal{S}_a.\ (\varphi(\mathcal{S}) = \mathcal{S}_a) \Longrightarrow \\ \mathcal{O}_{t\omega}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S})]\!] \subseteq \mathcal{O}_{t\omega}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), \mathcal{S}_a]\!]\,).$$

Theorem 2 says that linearizability with a progress property P together is equivalent to the corresponding contextual refinement $\sqsubseteq_{\varphi}^{P}$.

Theorem 2 (Equivalence). *$\Pi \preceq_{\varphi} \Pi_A \wedge P_{\varphi}(\Pi) \iff \Pi \sqsubseteq_{\varphi}^{P} \Pi_A$, where P is* wait-free*,* lock-free*,* obstruction-free*,* deadlock-free *or* starvation-free*.*

Here we assume the object specification Π_A is *total*, *i.e.*, the abstract operations never block. We provide the full proofs of our equivalence results in Appendix B.

The contextual refinement for wait-freedom takes $\mathcal{O}_{t\omega}$ at both the concrete and the abstract levels. The divergence of individual threads as well as I/O events are treated as observable behaviors. The intuition of the equivalence is as follows. Since a wait-free object Π guarantees that every method call finishes, we have to blame the client code itself for the divergence of a thread using Π. That is, even if the thread uses the abstract object Π_A, it must still diverge.

As an example, consider the client program (2.1). Intuitively, for any execution in which the client uses the abstract operations, only the right thread $\mathtt{t}_2$ diverges. Thus $\mathcal{O}_{t\omega}$ of the abstract program is a singleton set $\{(\epsilon, \{\mathtt{t}_2\})\}$. When the client uses the wait-free object in Figure 2(a), its $\mathcal{O}_{t\omega}$ set is still $\{(\epsilon, \{\mathtt{t}_2\})\}$. It does not produce more observable behaviors. But if it uses a non-wait-free object (such as the one in Figure 2(b)), the left thread $\mathtt{t}_1$ does not necessarily finish. The $\mathcal{O}_{t\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$. It produces more observable behaviors than the abstract client, breaking the contextual refinement. Thanks to observing div_tids that collects the diverging threads, we can rule out non-wait-free objects which may cause more threads to diverge.

$\Pi \sqsubseteq_{\varphi}^{\mathsf{lock\text{-}free}} \Pi_A$ takes coarser observable behaviors. We observe the divergence of the whole client program by using $\mathcal{O}_{\omega}$ at both the concrete and the abstract

levels. Intuitively, a lock-free object Π ensures that some method call will finish, thus the client using Π diverges only if there are an infinite number of method calls. Then it must also diverge when using the abstract object Π_A.

For example, consider the client (2.1). The whole client program diverges in every execution both when it uses the lock-free object in Figure 2(b) and when it uses the abstract one. The $\mathcal{O}_\omega$ set of observable behaviors is $\{\epsilon\}$ at both levels. On the other hand, the following client must terminate and print out both 1 and 2 in every execution. The $\mathcal{O}_\omega$ set is $\{1::2::\epsilon, 2::1::\epsilon\}$ at both levels.

```
inc(); print(1);    ||    dec(); print(2);
```
(5.3)

Instead, if the client (5.3) uses the non-lock-free object in Figure 2(c), it may diverge and nothing is printed out. The $\mathcal{O}_\omega$ set becomes $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, which contains more behaviors than the abstract side. Thus $\Pi \sqsubseteq_\varphi^{\text{lock-free}} \Pi_A$ fails.

Obstruction-freedom ensures progress for isolating executions in which eventually only one thread is running. Correspondingly, $\Pi \sqsubseteq_\varphi^{\text{obstruction-free}} \Pi_A$ restricts our considerations to isolating executions. It takes $\mathcal{O}_{i\omega}$ at the concrete level and $\mathcal{O}_\omega$ at the abstract level.

To understand the equivalence, consider the client (5.3) again. For isolating executions with the obstruction-free object in Figure 2(c), it *must* terminate and print out both 1 and 2. The $\mathcal{O}_{i\omega}$ set at the concrete level is $\{1::2::\epsilon, 2::1::\epsilon\}$, the same as the set $\mathcal{O}_\omega$ of the abstract side. Non-obstruction-free objects in general do not guarantee progress for some isolating executions. If the client uses the object in Figure 2(d) or (e), the $\mathcal{O}_{i\omega}$ set is $\{\epsilon, 1::2::\epsilon, 2::1::\epsilon\}$, not a subset of the abstract $\mathcal{O}_\omega$ set. The undesired empty observable trace is produced by unfair executions, where a thread acquires the lock and gets suspended and then the other thread would keep requesting the lock forever (it is executed in isolation).

$\Pi \sqsubseteq_\varphi^{\text{deadlock-free}} \Pi_A$ uses $\mathcal{O}_{f\omega}$ at the concrete side, ruling out undesired divergence caused by unfair scheduling. For the client (5.3) with the object in Figure 2(d) or (e), its $\mathcal{O}_{f\omega}$ set is same as the set $\mathcal{O}_\omega$ at the abstract level.

For $\Pi \sqsubseteq_\varphi^{\text{starvation-free}} \Pi_A$, we still consider only fair executions at the concrete level (similar to deadlock-freedom), but observe the divergence of individual threads rather than the whole program (similar to wait-freedom). It uses $\mathcal{O}_{ft\omega}$ at the concrete side and $\mathcal{O}_{t\omega}$ at the abstract level. For the client (5.3) with the starvation-free object in Figure 2(e), no thread diverges in any fair execution. Then the set $\mathcal{O}_{ft\omega}$ of observable behaviors is $\{(1::2::\epsilon, \emptyset), (2::1::\epsilon, \emptyset)\}$, which is same as the set $\mathcal{O}_{t\omega}$ at the abstract level.

Observing threaded divergence allows us to distinguish starvation-free objects from deadlock-free objects. Consider the client (2.1). Under fair scheduling, we know only the right thread $\mathtt{t}_2$ would diverge when using the starvation-free object in Figure 2(e). The set $\mathcal{O}_{ft\omega}$ is $\{(\epsilon, \{\mathtt{t}_2\})\}$. It coincides with the abstract behaviors $\mathcal{O}_{t\omega}$. But when using the deadlock-free object of Figure 2(d), the $\mathcal{O}_{ft\omega}$ set becomes $\{(\epsilon, \{\mathtt{t}_2\}), (\epsilon, \{\mathtt{t}_1, \mathtt{t}_2\})\}$, breaking the contextual refinement.

6 Related Work and Conclusion

There is a large body of work discussing the five progress properties and the contextual refinements individually. Our work in contrast studies their relationships, which have not been considered much before.

Gotsman and Yang [7] propose a new linearizability definition that preserves lock-freedom, and suggest a connection between lock-freedom and a termination-sensitive contextual refinement. We do not redefine linearizability here. Instead, we propose a unified framework to systematically relate all the five progress properties plus linearizability to various contextual refinements.

Herlihy and Shavit [11] informally discuss all the five progress properties. Our definitions in Section 4 mostly follow their explanations, but they are more formal and close the gap between program semantics and their history-based interpretations. We also notice that their obstruction-freedom is inappropriate for some examples (see TR [15]), and propose a different definition that is closer to the common intuition [10]. In addition, we relate the progress properties to contextual refinements, which consider the extensional effects on client behaviors.

Fossati *et al.* [5] propose a uniform approach in the π-calculus to formulate both the standard progress properties and their observational approximations. Their technical setting is completely different from ours. Also, their observational approximations for lock-freedom and wait-freedom are strictly weaker than the standard notions. Their deadlock-freedom and starvation-freedom are not formulated, and there is no observational approximation given for obstruction-freedom. In comparison, our framework relates each of the five progress properties (plus linearizablity) to an *equivalent* contextual refinement.

There are also formulations of progress properties based on temporal logics. For example, Petrank *et al.* [16] formalize the three non-blocking properties and Dongol [3] formalize all the five progress properties, using linear temporal logics. Those formulations make it easier to do model checking (*e.g.*, Petrank *et al.* [16] also build a tool to model check a variant of lock-freedom), while our contextual refinement framework is potentially helpful for modular Hoare-style verification.

Conclusion. We have introduced a contextual refinement framework to unify various progress properties. For linearizable objects, each progress property is equivalent to a specific termination-sensitive contextual refinement, as summarized in Table 1. The framework allows us to verify safety and liveness properties of client programs at a high abstraction level by replacing concrete method implementations with abstract operations. It also makes it possible to borrow ideas from existing proof methods for contextual refinements to verify linearizability and a progress property together, which we leave as future work.

Acknowledgments. We would like to thank anonymous referees for their helpful suggestions and comments. This work is supported in part by China Scholarship Council, NSFC grants 61073040 and 61229201, NCET grant NCET-2010-0984, and the Fundamental Research Funds for the Central Universities (Grant

No. WK0110000018). It is also supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0915888 and 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Aspnes, J., Herlihy, M.: Wait-free data structures in the asynchronous PRAM model. In: SPAA. pp. 340–349 (1990)
2. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: CSL. pp. 107–121 (2012)
3. Dongol, B.: Formalising progress properties of non-blocking programs. In: ICFEM. pp. 284–303 (2006)
4. Filipovic, I., O'Hearn, P., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. 411(51-52), 4379–4398 (2010)
5. Fossati, L., Honda, K., Yoshida, N.: Intensional and extensional characterisation of global progress in the π-calculus. In: CONCUR. pp. 287–301 (2012)
6. Gotsman, A., Yang, H.: Linearizability with ownership transfer. In: CONCUR'12
7. Gotsman, A., Yang, H.: Liveness-preserving atomicity abstraction. In: ICALP. pp. 453–465 (2011)
8. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
9. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: ICDCS. pp. 522–529 (2003)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (Apr 2008)
11. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS. pp. 313–328 (2011)
12. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
13. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: PLDI. p. to appear (2013)
14. Liang, H., Feng, X., Fu, M.: A rely-guarantee-based simulation for verifying concurrent program transformations. In: POPL. pp. 455–468 (2012)
15. Liang, H., Hoffmann, J., Feng, X., Shao, Z.: The extended version of the present paper (2013), `http://kyhcs.ustcsz.edu.cn/relconcur/prog`
16. Petrank, E., Musuvathi, M., Steensgaard, B.: Progress guarantee for parallel programs via bounded lock-freedom. In: PLDI. pp. 144–154 (2009)

A Comparisons with Herlihy and Shavit's Obstruction-Freedom

Herlihy and Shavit [11] define obstruction-freedom using the notion of *uniformly isolating* executions. A trace is uniformly isolating, if "for every $k > 0$, any thread that takes an infinite number of steps has an interval where it takes at least k concrete contiguous steps" [11]. Then, their obstruction-free object guarantees wait-freedom for every uniformly isolating execution. They also propose a new

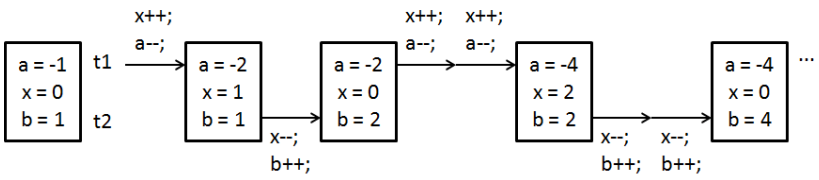


Fig. 10: Execution of `f()` || `g()` in Example 1

progress property, clash-freedom, which guarantees lock-freedom for uniformly-isolating executions.

Below we give an example showing that their definition is inconsistent with the common intuition of obstruction-freedom.

Example 1. The object implementation uses three shared variables: `x`, `a` and `b`. It provides two methods `f` and `g`:

```
f() {
  while (a <= x <= b) {
    x++;
    a--;
  }
}
```

```
g() {
  while (a <= x <= b) {
    x--;
    b++;
  }
}
```

We can see that, if `f()` or `g()` is eventually executed in isolation (*i.e.*, we suspend all but one threads), it must returns. Thus intuitively this object should be obstruction-free. It also satisfies our formulation (Definitions **??** and **??**).

However, we could construct an execution which is uniformly isolating but is not lock-free or wait-free. Consider the client program `f()` || `g()`. It has an execution shown in Figure 10. Starting from `x` $= 0$, `a` $= -1$ and `b` $= 1$, we alternatively let each thread execute more and more iterations. Then for any k, we could always find an interval of k iterations for each thread in this execution. Thus the execution is uniformly isolating. But neither method call finishes. This execution is not lock-free nor wait-free. Thus the object does not satisfy Herlihy and Shavit's obstruction-freedom or clash-freedom definitions.

B Proofs

In the following proofs, we make the call stacks explicit in the generation of event traces. For example, we use $\mathcal{H}[\![W, (\sigma_c, \sigma_o, \circledcirc)]\!]$ instead of $\mathcal{H}[\![W, (\sigma_c, \sigma_o)]\!]$. We generalize the definitions to allow nonempty call stacks in the initial state, *e.g.*, we can use $\mathcal{H}[\![W, (\sigma_c, \sigma_o, \mathcal{K})]\!]$.

B.1 Proofs of Theorem 1

To prove the theorem, we utilize the most general client (MGC). Let's assume $dom(\Pi) = \{f_1, \ldots, f_m\}$. We could use the expression **rand**() to get a random

(nondeterministic) integer, and $\mathbf{rand}(m)$ to get a random integer $r \in [1..m]$. Then, for any n, MGC_n is defined as follows:

$$\begin{array}{l}\mathsf{MGT} \stackrel{\text{def}}{=} \mathbf{while}\ (\mathbf{true})\{\ f_{\mathbf{rand}(m)}(\mathbf{rand}());\ \} \\ \mathsf{MGC}_n \stackrel{\text{def}}{=} \|_{i\in[1..n]}\,\mathsf{MGT}\end{array}$$

Here each thread keeps calling a random method with a random argument. We also define another kind of "most general clients" which print out arguments and return values for method calls:

$$\begin{array}{l}\mathsf{MGTp}_\mathsf{t} \stackrel{\text{def}}{=} \mathbf{while}\ (\mathbf{true})\{ \\ \qquad\qquad x_\mathsf{t} := \mathbf{rand}();\ y_\mathsf{t} := \mathbf{rand}(m);\ \mathbf{print}(y_\mathsf{t}, x_\mathsf{t}); \\ \qquad\qquad z_\mathsf{t} := f_{y_\mathsf{t}}(x_\mathsf{t});\ \mathbf{print}(z_\mathsf{t}); \\ \qquad\qquad \} \\ \mathsf{MGCp}_n \stackrel{\text{def}}{=} \|_{i\in[1..n]}\,\mathsf{MGTp}_i\end{array}$$

Here x_t, y_t and z_t are all local variables for thread t. Below we define the MGC versions of "linearizability" and refinements, and prove they are related to the standard definitions of linearizability and contextual refinement.

Definition 6. *$\Pi \preceq^{\mathsf{MGC}}_{\varphi} \Pi_A$ iff*

$$\begin{array}{l}\forall n, \sigma_o, \sigma_a, T.\ \ T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\varphi(\sigma_o) = \sigma_a) \\ \Longrightarrow\ \exists T_c, T_a.\ T_c \in \mathsf{completions}(T) \wedge \Pi_A \triangleright^{\mathsf{MGC}}_n (\sigma_a, T_a) \wedge T_c \preceq_{\mathsf{lin}} T_a\end{array}$$

where

$$\Pi_A \triangleright^{\mathsf{MGC}}_n (\sigma_a, T) \stackrel{\text{def}}{=} T \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!] \wedge \mathsf{seq}(T)\,.$$

$\Pi \subseteq_{\varphi} \Pi_A$ iff

$$\begin{array}{l}\forall n, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \\ \Longrightarrow\ \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

The following lemma shows that every history of an object Π could be generated by the MGC.

Lemma 1 (MGC is the Most General). *For any n, C_1, ..., C_n, σ_c, σ_o and σ_a, $\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$.*

Proof. We define the simulation relation $\precsim_{\mathsf{MGC}}$ between a program and a MGC in Figure 11(a), and prove the following (B.1) by case analysis and the operational semantics:

For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$ and e_1, if $(W_1, \mathcal{S}_1) \precsim_{\mathsf{MGC}} (W_2, \mathcal{S}_2)$, then

(1) if $(W_1, \mathcal{S}_1) \stackrel{e_1}{\longmapsto} \mathbf{abort}$ and $\mathsf{is\_obj\_abt}(e_1)$, then
there exists T_2 such that $(W_2, \mathcal{S}_2) \stackrel{T_2}{\longmapsto}{}^* \mathbf{abort}$ and
$e_1 = \mathsf{get\_hist}(T_2)$;

(2) if $(W_1, \mathcal{S}_1) \stackrel{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$, then
there exist T_2, W_2' and $\mathcal{S}_2'$ such that $(W_2, \mathcal{S}_2) \stackrel{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$,
$\mathsf{get\_hist}(e_1) = \mathsf{get\_hist}(T_2)$ and $(W_1', \mathcal{S}_1') \precsim_{\mathsf{MGC}} (W_2', \mathcal{S}_2')$.

(B.1)

$$(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\}))$$
$$\precsim_{\mathsf{MGC}} (\textbf{let } \Pi \textbf{ in } C'_1 \,\|\, \ldots \,\|\, C'_n, (\emptyset, \sigma_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\}))$$
$$\text{where } \forall i.\ (C_i, \kappa_i) \precsim_{\mathsf{MGC}} (C'_i, \kappa'_i)$$

$$(C, \circ) \precsim_{\mathsf{MGC}} (\mathsf{MGT}; \textbf{end}, \circ) \qquad (C, (\sigma_l, x, C')) \precsim_{\mathsf{MGC}} (C, (\sigma_l, \cdot, (\textbf{skip}; \mathsf{MGT}; \textbf{end})))$$

(a) Program is Simulated by MGC

$$(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\}))$$
$$\precsim_{\mathsf{MGCp}} (\textbf{let } \Pi \textbf{ in } C'_1 \,\|\, \ldots \,\|\, C'_n, (\sigma'_c, \sigma_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\}))$$
$$\text{where } \forall i.\ (C_i, \kappa_i) \precsim^{i}_{\mathsf{MGCp}} (C'_i, \kappa'_i) \text{ and } \sigma'_c = \{x_{\mathsf{t}} \rightsquigarrow \_, y_{\mathsf{t}} \rightsquigarrow \_, z_{\mathsf{t}} \rightsquigarrow \_ \mid 1 \le \mathsf{t} \le n\}$$

$$(C, \circ) \precsim^{\mathsf{t}}_{\mathsf{MGCp}} (\mathsf{MGTp}_{\mathsf{t}}; \textbf{end}, \circ)$$

$$(C, (\sigma_l, \cdot, C')) \precsim^{\mathsf{t}}_{\mathsf{MGCp}} (C, (\sigma_l, z_{\mathsf{t}}, (\textbf{skip}; \textbf{print}(z_{\mathsf{t}}); \mathsf{MGTp}_{\mathsf{t}}; \textbf{end})))$$

(b) Program is Simulated by MGCp

$$(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\}))$$
$$\precsim_{\mathsf{MGCp}^-} (\textbf{let } \Pi \textbf{ in } C'_1 \,\|\, \ldots \,\|\, C'_n, (\emptyset, \sigma_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\}))$$
$$\text{where } \forall i.\ (C_i, \sigma_c, \kappa_i) \precsim^{i,\Pi}_{\mathsf{MGCp}} (C'_i, \kappa'_i) \text{ and } \sigma_c = \{x_{\mathsf{t}} \rightsquigarrow \_, y_{\mathsf{t}} \rightsquigarrow \_, z_{\mathsf{t}} \rightsquigarrow \_ \mid 1 \le \mathsf{t} \le n\}$$

$$(C, \sigma_c, \circ) \precsim^{\mathsf{t},\Pi}_{\mathsf{MGCp}^-} \begin{cases} (C_o, (\{x \rightsquigarrow n\}, \cdot, (\textbf{skip}; \mathsf{MGT}; \textbf{end})) \\ \qquad \text{if } (C = \mathbf{E}[\, z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}})\,] \vee C = \mathbf{E}[\, \textbf{skip}; z_{\mathsf{t}} := f_{y_{\mathsf{t}}}(x_{\mathsf{t}})\,]) \\ \qquad\quad \wedge \sigma_c(x_{\mathsf{t}}) = n \wedge \sigma_c(y_{\mathsf{t}}) = i \wedge \Pi(f_i) = (x, C_o) \\ (\textbf{fret}(n'), (\_, \cdot, (\textbf{skip}; \mathsf{MGT}; \textbf{end})) \\ \qquad \text{if } (C = \mathbf{E}[\, \textbf{print}(z_{\mathsf{t}})\,] \vee C = \mathbf{E}[\, \textbf{skip}; \textbf{print}(z_{\mathsf{t}})\,]) \\ \qquad\quad \wedge \sigma_c(z_{\mathsf{t}}) = n' \\ (\mathsf{MGT}; \textbf{end}, \circ) \quad \text{otherwise} \end{cases}$$

$$(C, \sigma_c, (\sigma_l, z_{\mathsf{t}}, C')) \precsim^{\mathsf{t},\Pi}_{\mathsf{MGCp}^-} (C, (\sigma_l, \cdot, (\textbf{skip}; \mathsf{MGT}; \textbf{end})))$$

(c) MGCp is Simulated by MGC

$$(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\}))$$
$$\precsim (\textbf{let } \Pi_A \textbf{ in } C'_1 \,\|\, \ldots \,\|\, C'_n, (\emptyset, \sigma'_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\});$$
$$\quad \textbf{let } \Pi_A \textbf{ in } C''_1 \,\|\, \ldots \,\|\, C''_n, (\sigma_c, \sigma'_o, \{1 \rightsquigarrow \kappa''_1, \ldots, n \rightsquigarrow \kappa''_n\}))$$
$$\text{where } \forall i.\ (C_i, \kappa_i) \precsim (C'_i, \kappa'_i; C''_i, \kappa''_i)$$
$$\text{and } \mathcal{H}[\![\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\})]\!]$$
$$\subseteq \mathcal{H}[\![\textbf{let } \Pi_A \textbf{ in } C'_1 \,\|\, \ldots \,\|\, C'_n, (\emptyset, \sigma'_o, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\})]\!]$$

$$(C, \circ) \precsim (C', \circ; C, \circ) \qquad (C, (\sigma_l, x, C_c)) \precsim (C', (\sigma'_l, x', C'_c); C', (\sigma'_l, x, C_c))$$

(d) Concrete Program is Simulated by Abstract MGC and Abstract Program

Fig. 11: Simulations between Programs and MGC

With (B.1), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{H}[\![W_1, \mathcal{S}_1]\!]$.

$$\text{If } (\lfloor W_1 \rfloor, \mathcal{S}_1) \precsim_{\mathsf{MGC}} (\lfloor W_2 \rfloor, \mathcal{S}_2), \text{ then } \mathcal{H}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{H}[\![W_2, \mathcal{S}_2]\!].$$

Then, since

$$(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \precsim_{\mathsf{MGC}} (\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc)),$$

we are done. □

For linearizability, the MGC-version is equivalent to the original definition.

Lemma 2. $\Pi \preceq_\varphi \Pi_A \iff \Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$.

Proof. 1. $\Pi \preceq_\varphi \Pi_A \implies \Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$:
For any n, σ_o, σ_a and T such that $T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\varphi(\sigma_o) = \sigma_a$, from $\Pi \preceq_\varphi \Pi_A$, we know there exist T_c and T_a such that

$$T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd (\sigma_a, T_a) \wedge T_c \preceq_{\mathsf{lin}} T_a .$$

We only need to show that

$$\Pi_A \rhd (\sigma_a, T_a) \implies \Pi_A \rhd_n^{\mathsf{MGC}} (\sigma_a, T_a) .$$

First we know $\forall i.\ \mathsf{tid}(T_a(i)) \in [1..n]$. Second, from $\Pi_A \rhd (\sigma_a, T_a)$, we know there exist n', C_1, …, $C_{n'}$ and σ_c such that $\mathsf{seq}(T_a)$ and

$$T_a \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_{n'}), (\sigma_c, \sigma_a, \circledcirc)]\!] .$$

If $n' \leq n$, then we know

$$T_a \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_{n'} \,\|\, \mathbf{skip} \,\|\, \ldots \,\|\, \mathbf{skip}), (\sigma_c, \sigma_a, \circledcirc)]\!] .$$

From Lemma 1, we are done. Otherwise, since T_a only contains events of threads 1, …, n, we know the threads $n+1$, …, n' do not access the object. Similar to the proof of Lemma 1, we can construct simulations and prove $T_a \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$. Thus we are done.

2. $\Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A \implies \Pi \preceq_\varphi \Pi_A$:
For any n, C_1, …, C_n, σ_c, σ_o, σ_a and T such that $\varphi(\sigma_o) = \sigma_a$ and $T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, from Lemma 1, we know

$$T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] .$$

From $\Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$, we know there exist T_c and T_a such that

$$T_c \in \mathsf{completions}(T) \wedge \Pi_A \rhd_n^{\mathsf{MGC}} (\sigma_a, T_a) \wedge T_c \preceq_{\mathsf{lin}} T_a .$$

By definitions, we see

$$\Pi_A \rhd_n^{\mathsf{MGC}} (\sigma_a, T_a) \implies \Pi_A \rhd (\sigma_a, T_a) .$$

Thus we are done. □

Below we prove an important lemma which relates the basic contextual refinement to a refinement over MGC which considers histories instead of observable behaviors. The idea behind this lemma will be useful in proving various equivalence results, including those for progress properties.

Lemma 3. $\Pi \sqsubseteq_\varphi \Pi_A \iff \Pi \subseteq_\varphi \Pi_A$.

Proof. 1. $\Pi \sqsubseteq_\varphi \Pi_A \implies \Pi \subseteq_\varphi \Pi_A$:

We first prove the following (a) and (b):

(a) For any n, σ_o, σ_c, T,
if $\sigma_c = \{x_\mathsf{t} \rightsquigarrow \_, y_\mathsf{t} \rightsquigarrow \_, z_\mathsf{t} \rightsquigarrow \_ \mid 1 \le \mathsf{t} \le n\}$ and
$T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$,
then there exists $\mathcal{B}$ such that $T \approx \mathcal{B}$ and
$\mathcal{B} \in \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp}_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$,
where

$$\frac{}{\epsilon \approx \epsilon} \qquad \frac{\lambda \approx e \qquad T \approx \mathcal{B}}{\lambda :: T \approx e :: \mathcal{B}}$$

$$\frac{}{(\mathsf{t}, f_i, n) \approx (\mathsf{t}, \mathbf{out}, (i, n))} \qquad \frac{}{(\mathsf{t}, \mathbf{ret}, n) \approx (\mathsf{t}, \mathbf{out}, n)}$$

$$\frac{}{(\mathsf{t}, \mathbf{obj}, \mathbf{abort}) \approx (\mathsf{t}, \mathbf{obj}, \mathbf{abort})}$$

Proof. We define the simulation relation $\precsim_{\mathsf{MGCp}}$ in Figure 11(b), and prove the following (B.2) by case analysis and the operational semantics. This simulation ensures that at the right side (MGCp), each output of the method argument is immediately followed by invoking the method, and each method return is immediately followed by printing out the return value.

For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$ and e_1, if $(W_1, \mathcal{S}_1) \precsim_{\mathsf{MGCp}} (W_2, \mathcal{S}_2)$, then
(1) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} \mathbf{abort}$ and $\mathsf{is\_obj\_abt}(e_1)$, then
there exists T_2 such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* \mathbf{abort}$ and
$e_1 \approx \mathsf{get\_obsv}(T_2)$;
(2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$, then
there exist T_2, W_2' and $\mathcal{S}_2'$ such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$,
$\mathsf{get\_hist}(e_1) \approx \mathsf{get\_obsv}(T_2)$ and $(W_1', \mathcal{S}_1') \precsim_{\mathsf{MGCp}} (W_2', \mathcal{S}_2')$.
(B.2)

With (B.2), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{H}[\![W_1, \mathcal{S}_1]\!]$.

If $(\lfloor W_1 \rfloor, \mathcal{S}_1) \precsim_{\mathsf{MGCp}} (\lfloor W_2 \rfloor, \mathcal{S}_2)$ and $T \in \mathcal{H}[\![W_1, \mathcal{S}_1]\!]$, then
there exists $\mathcal{B}$ such that $T \approx \mathcal{B}$ and $\mathcal{B} \in \mathcal{O}[\![W_2, \mathcal{S}_2]\!]$.

Then, since

$$(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc)) \precsim_{\mathsf{MGCp}} (\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp}_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)),$$

we are done.

(b) For any n, σ_a, σ_c, $\mathcal{B}$,
if $\sigma_c = \{x_\mathsf{t} \rightsquigarrow \_, y_\mathsf{t} \rightsquigarrow \_, z_\mathsf{t} \rightsquigarrow \_ \mid 1 \le \mathsf{t} \le n\}$ and
$\mathcal{B} \in \mathcal{O}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp}_n), (\sigma_c, \sigma_a, \circledcirc)]\!]$,
then there exists T such that $T \approx \mathcal{B}$ and
$T \in \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$.

Proof. We define the simulation relation $\precsim_{\mathsf{MGCp}^-}$ in Figure 11(c), and prove the following (B.3) by case analysis and the operational semantics. This simulation ensures two things. (i) Whenever the left side (MGCp) prints out a method argument, the right side (MGC) invokes the method using that argument. (ii) Whenever the left side prints out a return value, the right side must return the same value. We can ensure (i) and (ii) because x_t, y_t and z_t are all thread-local variables.

For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$ and e_1, if $(W_1, \mathcal{S}_1) \precsim_{\mathsf{MGCp}^-} (W_2, \mathcal{S}_2)$, then
(1) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} \textbf{abort}$, then
there exists T_2 such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* \textbf{abort}$ and
$\mathsf{get\_hist}(T_2) \approx e_1$;
(2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$, then
there exist T_2, W_2' and $\mathcal{S}_2'$ such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$,
$\mathsf{get\_hist}(T_2) \approx \mathsf{get\_obsv}(e_1)$ and $(W_1', \mathcal{S}_1') \precsim_{\mathsf{MGCp}^-} (W_2', \mathcal{S}_2')$. (B.3)

With (B.3), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{O}[\![W_1, \mathcal{S}_1]\!]$.
If $(\lfloor W_1 \rfloor, \mathcal{S}_1) \precsim_{\mathsf{MGCp}^-} (\lfloor W_2 \rfloor, \mathcal{S}_2)$ and $\mathcal{B} \in \mathcal{O}[\![W_1, \mathcal{S}_1]\!]$, then there exists T such that $T \approx \mathcal{B}$ and $T \in \mathcal{H}[\![W_2, \mathcal{S}_2]\!]$.
Then, since

$$(\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGCp}_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \precsim_{\mathsf{MGCp}^-} (\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_a, \circledcirc)),$$

we are done.

Then, since $\Pi \sqsubseteq_\varphi \Pi_A$, we know

$$\forall n, \sigma_c, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \implies \mathcal{O}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp}_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGCp}_n), (\sigma_c, \sigma_a, \circledcirc)]\!].$$

Thus from (a) and (b), we get

$$\forall n, \sigma_o, \sigma_a.\ (\varphi(\sigma_o) = \sigma_a) \implies \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!].$$

Then we are done.

2. $\Pi \subseteq_\varphi \Pi_A \implies \Pi \sqsubseteq_\varphi \Pi_A$:
We define the simulation relation $\precsim$ in Figure 11(d), and prove the following (B.4) by case analysis and the operational semantics. This simulation relates one program to two programs. We use the MGC at the abstract level to help determine the abstract program that corresponds to the concrete one.

Specifically, we require the histories generated by the concrete program can also be generated by the abstract MGC. Then, when an abstract thread is in a method call, its code should be the same as the MGC thread. Otherwise, its code is the same as the concrete thread.

For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$ and e_1,
if $(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, then
(1) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto}$ **abort**, then
there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^*$ **abort** and
$e_1 = \mathsf{get\_obsv}(T_3)$;
(2) if $(W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1')$, then
there exist T_2, W_2', $\mathcal{S}_2'$, T_3, W_3' and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^* (W_3', \mathcal{S}_3')$,
$\mathsf{get\_obsv}(e_1) = \mathsf{get\_obsv}(T_3)$ and $(W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.
(B.4)

With (B.4), we can prove the following by induction over the number of steps generating the event trace of $\mathcal{O}[\![W_1, \mathcal{S}_1]\!]$.

If $(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, then $\mathcal{O}[\![W_1, \mathcal{S}_1]\!] \subseteq \mathcal{O}[\![W_3, \mathcal{S}_3]\!]$.
For any n, C_1, …, C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] .$$

Since $\Pi \subseteq_\varphi \Pi_A$, we know if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!] .$$

Then we know

$$\begin{array}{l}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc); \\ \quad \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)) .\end{array}$$

Thus, we get

$$\begin{array}{l}\mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!] .\end{array}$$

Thus we are done. □

Then, we prove the following (B.5) and can get Theorem 1.

$$\Pi \subseteq_\varphi \Pi_A \iff \Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A \tag{B.5}$$

1. $\Pi \subseteq_\varphi \Pi_A \implies \Pi \preceq_\varphi^{\mathsf{MGC}} \Pi_A$:
We only need to prove the following lemma (remember we assume that each C_i in Π_A is of the form $\langle C \rangle$ and it is always safe to execute Π_A).

Lemma 4 (Π_A is Linearizable). *For any n, σ_a and T,*
if $T \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$,
then there exist T_c and T_a such that $T_c \in \mathsf{completions}(T)$, $T_c \preceq_{\mathsf{lin}} T_a$,
$T_a \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$ and $\mathsf{seq}(T_a)$.

Proof. We define a new operational semantics, in which we additionally generate two events at the single step of the method body. We know the method body in the execution can only be $\langle C\rangle; \mathbf{noret}$, and hence the resulting code after one step (if not block) must be $\mathbf{fret}(n')$ for some n'.

$$\frac{(\langle C\rangle; \mathbf{noret}, \sigma_o \uplus \sigma_l) \longrightarrow_{\mathsf{t}} (\mathbf{fret}(n'), \sigma_o' \uplus \sigma_l') \quad dom(\sigma_l) = dom(\sigma_l') \qquad \sigma_l = \{y \rightsquigarrow n\} \qquad \Pi(f) = (y, \langle C\rangle)}{(\langle C\rangle; \mathbf{noret}, (\sigma_c, \sigma_o, (\sigma_l, x, C_c))) \xrightarrow{[\mathsf{t},f,n]::[\mathsf{t},\mathbf{ret},n']}_{\mathsf{t},\Pi} (\mathbf{fret}(n'), (\sigma_c, \sigma_o', (\sigma_l', x, C_c)))}$$

Here $[\mathsf{t}, f, n]$ and $[\mathsf{t}, \mathbf{ret}, n']$ are two new events (called *atom-invocation event* and *atom-return event* respectively) generated for the new semantics. We use $T|_{[]}$ to project the event trace T to the new events, and use $\lfloor e \rfloor$ (and $\lfloor T \rfloor$) to transform the new event (and the event trace) to an old event (and a trace of old events), where $[\mathsf{t}, f, n]$ is transformed to (t, f, n) and $[\mathsf{t}, \mathbf{ret}, n']$ is transformed to $(\mathsf{t}, \mathbf{ret}, n')$. Other parts of the semantics are the same as the operational semantics in Figure 5. We can define $\mathcal{T}_{[]}[\![W, \mathcal{S}]\!]$ in a similar way as $\mathcal{T}[\![W, \mathcal{S}]\!]$, which uses the new semantics instead of the original one and keeps all the events including the new events.

(1) We can prove that there is a lock-step simulation between the original semantics in Figure 5 and the new semantics. Then, for any T such that
$$T \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!],$$
we have an corresponding execution under the new semantics to generate T_T such that
$$T_T \in \mathcal{T}_{[]}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!],$$
and $\mathsf{get\_hist}(T_T) = T$.

(2) Below we show:
If $T_T \in \mathcal{T}_{[]}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$, $T = \mathsf{get\_hist}(T_T)$ and $T_a = \lfloor T_T|_{[]} \rfloor$,
then $\mathsf{seq}(T_a)$ and there exists T_c such that $T_c \in \mathsf{completions}(T)$ and $T_c \preceq_{\mathsf{lin}} T_a$.

Proof. By the new operational semantics, we know $\mathsf{seq}(T_a)$ holds.

Construct T_c and Prove Linearizability Condition 1: By the new operational semantics, we know that for any t, $T|_{\mathsf{t}}$ and $T_a|_{\mathsf{t}}$ must satisfy one of the following:
(i) $T|_{\mathsf{t}} = T_a|_{\mathsf{t}}$; or
(ii) $\exists n.\ T|_{\mathsf{t}} :: (\mathsf{t}, \mathbf{ret}, n) = T_a|_{\mathsf{t}}$; or
(iii) $\exists f, n.\ T|_{\mathsf{t}} = T_a|_{\mathsf{t}} :: (\mathsf{t}, f, n)$.
We construct T_e as follows. For any t, if it is the above case (ii), we append the corresponding return event at the end of T. Since $\mathsf{well\_formed}(T)$ and $\mathsf{well\_formed}(T_a)$, we could prove $\mathsf{well\_formed}(T_e)$. Thus $T_e \in \mathsf{extensions}(T)$.
Also T_e satisfies: for any t, one of the following holds:
(i) $T_e|_{\mathsf{t}} = T_a|_{\mathsf{t}}$; or
(ii) $\exists f, n.\ T_e|_{\mathsf{t}} = T_a|_{\mathsf{t}} :: (\mathsf{t}, f, n)$.
Let $T_c = \mathsf{truncate}(T_e)$. Thus $T_c \in \mathsf{completions}(T)$.
Since $\forall \mathsf{t}.\ \mathsf{is\_res}(\mathsf{last}(T_a|_{\mathsf{t}})) \wedge \mathsf{seq}(T_a|_{\mathsf{t}})$, we could prove that for any t,
(i) if $T_e|_{\mathsf{t}} = T_a|_{\mathsf{t}}$, then $T_c|_{\mathsf{t}} = T_e|_{\mathsf{t}}$;
(ii) if $T_e|_{\mathsf{t}} = T_a|_{\mathsf{t}} :: (\mathsf{t}, f, n)$, then $T_c|_{\mathsf{t}} = T_a|_{\mathsf{t}}$.
Thus $\forall \mathsf{t}.\ T_c|_{\mathsf{t}} = T_a|_{\mathsf{t}}$.

Prove Linearizability Condition 2: We informally show that the bijection π implicit in $\forall \mathsf{t}.\ T_c|_{\mathsf{t}} = T_a|_{\mathsf{t}}$ preserves the response-invocation order.
Let $T_c(i)$ be a response event in T_c and let $T_c(j)$ be an invocation event. Then $\pi(i)$ and $\pi(j)$ are the indices of $T_c(i)$ and $T_c(j)$ in T_a respectively. Suppose $i < j$. By the construction of T_c from T, we know the same response and invocation events are in T, and the response happens before the invocation. Let i' and j' be the indices of these events in T. Then $i' < j'$. By the new operational semantics, we know in T_T, the atom-return event is before the atom-invocation event since the history return event is before the history invocation event. Thus $\pi(i) < \pi(j)$.

(3) Finally, we show the following and finish the proof of the lemma:
If $T_T \in \mathcal{T}_{[]}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$ and $T_a = \lfloor T_T|_{[]} \rfloor$,
then $T_a \in \mathcal{H}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$.
This is proved by constructing the following simulation $\precsim_{\mathsf{new}}$. This simulation ensures that the right side invokes and returns from a method at the time when the left side generates the new atomic events.

$$(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\emptyset, \sigma_a, \{1 \rightsquigarrow \kappa_1, \ldots, n \rightsquigarrow \kappa_n\}))$$
$$\precsim_{\mathsf{new}} (\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ C'_1 \,\|\, \ldots \,\|\, C'_n, (\emptyset, \sigma_a, \{1 \rightsquigarrow \kappa'_1, \ldots, n \rightsquigarrow \kappa'_n\})) \quad \text{where } \forall i.\ (C_i, \kappa_i) \precsim_{\mathsf{new}} (C'_i, \kappa'_i)$$

$$(C, \circ) \precsim_{\mathsf{new}} (C, \circ)$$

$$((\langle C \rangle; \mathbf{noret}),\ (\sigma_l, \cdot, (\mathbf{skip}; \mathsf{MGT}))) \precsim_{\mathsf{new}} ((f_{\mathbf{rand}(m)}(\mathbf{rand}()); \mathsf{MGT}),\ \circ)$$

$$(\mathbf{fret}(n'), (\sigma_l, \cdot, (\mathbf{skip}; \mathsf{MGT}))) \precsim_{\mathsf{new}} ((\mathbf{skip}; \mathsf{MGT}), \circ)$$

We prove the following by case analysis and the operational semantics.
For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$ and T_1,
if $(W_1, \mathcal{S}_1) \precsim_{\mathsf{new}} (W_2, \mathcal{S}_2)$ and $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto} (W'_1, \mathcal{S}'_1)$ in the new semantics,
then there exist T_2, W'_2 and $\mathcal{S}'_2$ such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W'_2, \mathcal{S}'_2)$, $\mathsf{get\_hist}(T_2) = \lfloor T_1|_{[]} \rfloor$ and $(W'_1, \mathcal{S}'_1) \precsim_{\mathsf{new}} (W'_2, \mathcal{S}'_2)$.
Then we can prove the following by induction over the number of steps generating the event trace of $\mathcal{T}_{[]}[\![W_1, \mathcal{S}_1]\!]$.
If $(W_1, \mathcal{S}_1) \precsim_{\mathsf{new}} (W_2, \mathcal{S}_2)$, $T_T \in \mathcal{T}_{[]}[\![W_1, \mathcal{S}_1]\!]$ and $T_a = \lfloor T_T|_{[]} \rfloor$,
then $T_a \in \mathcal{H}[\![W_2, \mathcal{S}_2]\!]$.
Since we know
$$(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)) \precsim_{\mathsf{new}} (\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)),$$
we are done.

The lemma is immediate from the above (1), (2) and (3). □

2. $\mathit{\Pi} \preceq_{\varphi}^{\mathsf{MGC}} \mathit{\Pi}_A \implies \mathit{\Pi} \subseteq_{\varphi} \mathit{\Pi}_A$:
We only need to prove the following lemma (similar to the Rearrangement Lemma in [6]):

Lemma 5 (Rearrangement). *For any n, σ_a, T and T_a,*
if $T \preceq_{\mathsf{lin}} T_a$, $T_a \in \mathcal{H}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$ and $\mathsf{seq}(T_a)$,
then $T \in \mathcal{H}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$.

Proof. Suppose $|T| = n$. We know T must not contain the abort event. From $T \preceq_{\mathsf{lin}} T_a$, we know
(i) $\forall \mathsf{t}.\ T|_{\mathsf{t}} = T_a|_{\mathsf{t}}$;
(ii) there exists a bijection $\pi : \{1, \ldots, |T|\} \to \{1, \ldots, |T_a|\}$ such that $\forall i.\, T(i) = T_a(\pi(i))$ and $\forall i, j.\ i < j \wedge \mathsf{is\_res}(T(i)) \wedge \mathsf{is\_inv}(T(j)) \implies \pi(i) < \pi(j)$.
We construct the execution under the new semantics (defined in the proof of Lemma 4) which generates T, and the new events constitute T_a, *i.e.*, we want to show the following holds:

$$\exists T_T.\ T_T \in \mathcal{T}_{[]}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!] \wedge T = \mathsf{get\_hist}(T_T)\,. \quad \text{(B.6)}$$

Then we prove that there is a lock-step simulation between the new semantics and the original semantics in Figure 5, and we can get

$$T \in \mathcal{H}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!].$$

Below we prove (B.6). We prove that for any k, there exist T_T, W', $\mathcal{S}'$ and k' such that

$$\begin{array}{l}(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)) \overset{T_T}{\longmapsto}{}^{*}\ (W', \mathcal{S}') \\ \wedge\, \mathsf{get\_hist}(T_T) = T(1..k) \wedge \lfloor T_T|_{[]} \rfloor = T_a(1..k') \\ \wedge\, (\forall \mathcal{S}''.\ (\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)) \overset{T_a(1..k')}{\longmapsto}{}^{*}\ (\_, \mathcal{S}'') \\ \quad \implies \mathcal{S}''|_{\mathbf{obj}} = \mathcal{S}'|_{\mathbf{obj}})\,,\end{array}$$

where $\mathcal{S}'|_{\mathbf{obj}}$ get the object memory in $\mathcal{S}'$.
By induction over k.
Base Case: If $k = 0$, trivial.
Inductive Step: Suppose there exist T_1, W_1, $\mathcal{S}_1$ and k_1 such that

$$\begin{array}{l}(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)) \overset{T_1}{\longmapsto}{}^{*}\ (W_1, \mathcal{S}_1) \\ \wedge\, \mathsf{get\_hist}(T_1) = T(1..k) \wedge \lfloor T_1|_{[]} \rfloor = T_a(1..k_1) \\ \wedge\, (\forall \mathcal{S}_1'.\ (\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)) \overset{T_a(1..k_1)}{\longmapsto}{}^{*}\ (\_, \mathcal{S}_1') \\ \quad \implies \mathcal{S}_1'|_{\mathbf{obj}} = \mathcal{S}_1|_{\mathbf{obj}})\,,\end{array}$$

we want to show there exist T_2, W_2, $\mathcal{S}_2$ and k_2 such that

$$\begin{array}{l}(W_1, \mathcal{S}_1) \overset{T_2}{\longmapsto}{}^{*}\ (W_2, \mathcal{S}_2) \\ \wedge\, \mathsf{get\_hist}(T_2) = T(k+1) \wedge \lfloor T_2|_{[]} \rfloor = T_a(k_1 + 1..k_2) \\ \wedge\, (\forall \mathcal{S}_2'.\ (\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc)) \overset{T_a(1..k_2)}{\longmapsto}{}^{*}\ (\_, \mathcal{S}_2') \\ \quad \implies \mathcal{S}_2'|_{\mathbf{obj}} = \mathcal{S}_2|_{\mathbf{obj}})\,,\end{array}$$

By case analysis.
(a) $T(k+1) = (\mathsf{t}, f, n')$.
Suppose $T(k+1) = (T|_{\mathsf{t}})(i)$.
From $T|_{\mathsf{t}} = T_a|_{\mathsf{t}}$ and $T_a \in \mathcal{H}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$, we know $i = 1$ or $\mathsf{is\_ret}((T|_{\mathsf{t}})(i-1))$ holds.
i. If $i = 1$, we just let the code MGT of the thread t executes to calling the method f using the argument n, and generates the event (t, f, n').

ii. If $\mathsf{is\_ret}((T|_{\mathsf{t}})(i-1))$ holds, we know the code of the thread t is in the client code. Still we can let it execute to the method call of f using the argument n, generating the event (t, f, n').

(b) $T(k+1) = (\mathsf{t}, \mathbf{ret}, n')$.

Suppose $T(k+1) = (T|_{\mathsf{t}})(i)$. Similar to the previous case, we know $\mathsf{is\_inv}((T|_{\mathsf{t}})(i-1))$ holds. Suppose $(T|_{\mathsf{t}})(i-1) = e = (\mathsf{t}, f, n)$ and $\Pi_A(f) = (x, \langle C \rangle)$. Thus the code of the thread t is either $\langle C \rangle; \mathbf{noret}$ or $\mathbf{fret}(n'')$ (for some n'').

i. The code of t is $\langle C \rangle; \mathbf{noret}$.

Thus $\mathsf{last}(T_1|_{\mathsf{t}}) = e$. Suppose $|T(1..k)|_{\mathsf{t}}| = n_1$. From the operational semantics and the generation of T_1, we know $|T_a(1..k_1)|_{\mathsf{t}}| = n_1 - 1$. For the bijection π in (ii) which maps events in T to events of T_a, we let $k_2 = \pi(k+1)$. Since $T|_{\mathsf{t}} = T_a|_{\mathsf{t}}$, we know $k_2 > k_1$. Let $k' = k_2 - k_1$. Suppose $T_a(k_1 + 1..k_2) = e_1 :: \ldots :: e_{k'}$. Since $\lfloor T_1|_{[]} \rfloor = T_a(1..k_1)$, by the operational semantics and the generation of T_1, we know $\mathsf{is\_ret}(T_a(k_1))$. Since $\mathsf{seq}(T_a)$, we know $\mathsf{seq}(e_1 :: \ldots :: e_{k'})$ and $k' = 2j$. Suppose the threads of the events $e_1, \ldots, e_{k'}$ are $\mathsf{t}_1, \ldots, \mathsf{t}_j$ respectively where $\mathsf{t}_j = \mathsf{t}$. Below we prove that for any i such that $1 \leq i \leq j$, the current code of the thread t_i is $\langle C_i \rangle; \mathbf{noret}$ (for some method body $\langle C_i \rangle$), and $e_{2i-1} = \mathsf{last}(T(1..k)|_{\mathsf{t}_i})$. The proof is by contradiction. Suppose $e_{2i-1} = T(i')$ and $i' > k$. Since $T(k+1) = (\mathsf{t}, \mathbf{ret}, n')$ and $\mathsf{is\_inv}(e_{2i-1})$, we know $i' > k+1$. By (ii), we know $\pi(i') > \pi(k+1) = k_2$, which contradicts the fact that e_{2i-1} is an event in $T_a(k_1 + 1..k_2)$. Thus, $i' \leq k$, and since $\lfloor T_1|_{[]} \rfloor = T_a(1..k_1)$, by the operational semantics and the generation of T_1, we know $e_{2i-1} = \mathsf{last}(T_1|_{\mathsf{t}_i})$. Thus we are done.

We let the threads $\mathsf{t}_1, \ldots, \mathsf{t}_j$ execute one step in order, generating the event trace T_2' which only contains the atom-invocation and atom-return events, and then the thread t_j execute one more step generating $e_{k'} = T_a(k_2) = T_a(\pi(k+1)) = T(k+1)$. Since $T_a \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$, we know this execution is possible, and moreover we have $\lfloor T_2|_{[]} \rfloor = \lfloor T_2' \rfloor = T_a(k_1 + 1..k_2)$.

ii. The code of t is $\mathbf{fret}(n'')$.

Thus $\mathsf{last}(T_1|_{\mathsf{t}}) = [\mathsf{t}, \mathbf{ret}, n'']$. Since $\lfloor T_1|_{[]} \rfloor = T_a(1..k_1)$, we know $\mathsf{last}(T_a(1..k_1)|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{ret}, n'')$. Suppose $|T_a(1..k_1)|_{\mathsf{t}}| = n_1$.

From the operational semantics and the generation of T_1, we know $|\mathsf{get\_hist}(T_1|_{\mathsf{t}})| = |T(1..k)|_{\mathsf{t}}| = n_1 - 1$. Since $T|_{\mathsf{t}} = T_a|_{\mathsf{t}}$, we know $n' = n''$. The code of t is $\mathbf{fret}(n')$. We let it execute one step and generate the event $(\mathsf{t}, \mathbf{ret}, n')$.

Thus (B.6) holds and we are done. □

From $\Pi \preceq_{\varphi}^{\mathsf{MGC}} \Pi_A$, we know

$$\begin{aligned}&\forall n, \sigma_o, \sigma_a, T.\ \ T \in \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\varphi(\sigma_o) = \sigma_a)\\ \Longrightarrow\ &\exists T_c, T_a.\ T_c \in \mathsf{completions}(T) \wedge T_a \in \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\\ &\qquad \wedge \mathsf{seq}(T_a) \wedge T_c \preceq_{\mathsf{lin}} T_a\end{aligned}$$

From Lemma 5, we know

$$\forall n, \sigma_o, \sigma_a, T.\ \ T \in \mathcal{H}[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\varphi(\sigma_o) = \sigma_a)$$
$$\Longrightarrow\ \exists T_c.\ T_c \in \mathsf{completions}(T) \wedge T_c \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$$

Since $T_c \in \mathsf{completions}(T)$, we know there exists T_e such that $T_c = \mathsf{truncate}(T_e)$ and $T_e \in \mathsf{extensions}(T)$. By the definition of $\mathsf{truncate}(T_e)$, we can prove:

$$T_e \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$$

Then, by the definition of $T_e \in \mathsf{extensions}(T)$, we can prove:

$$T \in \mathcal{H}[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$$

Thus we get $\Pi \subseteq_{\varphi} \Pi_A$.

B.2 Proofs of Figures 1 and 7

Lemma 6 (Figure 7). *Assume* $T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$.

1. $\mathsf{wait\text{-}free}(T) \iff \mathsf{prog\text{-}t}(T) \vee \mathsf{non\text{-}sched}(T) \vee \mathsf{abt}(T) \iff \mathsf{non\text{-}sched}(T) \vee \mathsf{abt}(T)$;
2. $\mathsf{lock\text{-}free}(T) \iff \mathsf{prog\text{-}s}(T) \vee \mathsf{non\text{-}sched}(T) \vee \mathsf{abt}(T) \iff \mathsf{wait\text{-}free}(T) \vee \mathsf{prog\text{-}s}(T)$;
3. $\mathsf{obstruction\text{-}free}(T) \iff \mathsf{prog\text{-}t}(T) \vee \mathsf{non\text{-}sched}(T) \vee \neg\mathsf{iso}(T) \vee \mathsf{abt}(T) \iff \mathsf{lock\text{-}free}(T) \vee \neg\mathsf{iso}(T)$;
4. $\mathsf{deadlock\text{-}free}(T) \iff \mathsf{prog\text{-}s}(T) \vee \neg\mathsf{fair}(T) \vee \mathsf{abt}(T) \iff \mathsf{lock\text{-}free}(T) \vee \neg\mathsf{fair}(T)$;
5. $\mathsf{starvation\text{-}free}(T) \iff \mathsf{prog\text{-}t}(T) \vee \neg\mathsf{fair}(T) \vee \mathsf{abt}(T) \iff \mathsf{wait\text{-}free}(T) \vee \neg\mathsf{fair}(T)$.

Proof. 1. By definition.

$$\begin{aligned}
\mathsf{wait\text{-}free}(T) &\iff (\,\forall i, e.\ e \in \mathsf{pend_inv}(T(1..i)) \\
&\qquad \Longrightarrow\ (\exists j.\ j > i \wedge \mathsf{match}(e, T(j))) \\
&\qquad\quad \vee (\exists j.\ j > i \wedge (\forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)))\,) \\
&\qquad \vee \mathsf{abt}(T) \\
&\iff (\,\forall i, e.\ e \in \mathsf{pend_inv}(T(1..i)) \wedge \neg(\exists j.\ j > i \wedge \mathsf{match}(e, T(j))) \\
&\qquad \Longrightarrow\ (\exists j.\ j > i \wedge (\forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)))\,) \\
&\qquad \vee \mathsf{abt}(T) \\
&\iff (\,\forall e.\ e \in \mathsf{pend_inv}(T)\ \Longrightarrow\ (\exists j.\ \forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{tid}(e))\,) \\
&\qquad \vee \mathsf{abt}(T) \\
&\iff \mathsf{non\text{-}sched}(T) \vee \mathsf{abt}(T)
\end{aligned}$$

Also, we can prove $\mathsf{prog\text{-}t}(T) \Longrightarrow \mathsf{non\text{-}sched}(T)$ as follows.

$$\begin{aligned}
\mathsf{prog\text{-}t}(T) &\iff (\,\forall i, e.\ e \in \mathsf{pend_inv}(T(1..i))\ \Longrightarrow\ \exists j.\ j > i \wedge \mathsf{match}(e, T(j))\,) \\
&\iff (\,\forall i, e.\ e \in \mathsf{pend_inv}(T(1..i))\ \Longrightarrow\ e \notin \mathsf{pend_inv}(T)\,) \\
&\iff (\,\mathsf{pend_inv}(T) = \emptyset\,) \\
&\Longrightarrow \mathsf{non\text{-}sched}(T)
\end{aligned}$$

2. We only need to prove the first equivalence. The second is trivial from the first one.

$$\begin{aligned}\mathsf{lock\text{-}free}(T) \iff & (\,\forall i, e.\ e \in \mathsf{pend\_inv}(T(1..i)) \\ & \quad \implies (\exists j.\ j > i \wedge \mathsf{is\_ret}(T(j))) \\ & \qquad \vee (\exists j.\ j > i \wedge (\forall k \geq j.\ \mathsf{is\_clt}(T(k))))\,) \\ & \vee \mathsf{abt}(T) \\ \iff & \mathsf{prog\text{-}s}(T) \\ & \vee (\,\exists j.\ \forall k \geq j.\ \mathsf{is\_clt}(T(k))\,) \\ & \vee \mathsf{abt}(T)\end{aligned}$$

From $\exists j.\ \forall k \geq j.\ \mathsf{is\_clt}(T(k))$ and the operational semantics generating T, we know $\mathsf{non\text{-}sched}(T)$ holds.
If $\mathsf{non\text{-}sched}(T)$ holds, we know there exists j such that $\forall k \geq j.\ \mathsf{tid}(T(k)) \notin \mathsf{tid}(\mathsf{pend\_inv}(T))$, where $\mathsf{tid}(\mathsf{pend\_inv}(T))$ gets the set of thread IDs of the pending invocations in T. Then by the operational semantics and the generation of T, we know either $\exists j.\ \forall k \geq j.\ \mathsf{is\_clt}(T(k))$ or $\mathsf{prog\text{-}s}(T)$ holds.

3. For obstruction-freedom, we only need to prove the following:
(1) $\forall T.\ \mathsf{iso}(T) \wedge \mathsf{obstruction\text{-}free}(T) \implies \mathsf{wait\text{-}free}(T)$;
(2) $\forall T.\ \mathsf{wait\text{-}free}(T) \implies \mathsf{obstruction\text{-}free}(T)$;
(3) $\forall T.\ \neg\mathsf{iso}(T) \implies \mathsf{obstruction\text{-}free}(T)$;
(4) $\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T.$
$T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \wedge \mathsf{prog\text{-}s}(T)$
$\implies \mathsf{obstruction\text{-}free}(T)$.
For (1) $\forall T.\ \mathsf{iso}(T) \wedge \mathsf{obstruction\text{-}free}(T) \implies \mathsf{wait\text{-}free}(T)$:

Proof. By $\mathsf{obstruction\text{-}free}(T)$, we know one of the following holds:
(a) there exists i such that $\mathsf{is\_abt}(T(i))$ holds; or
(b) for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then one of the following holds:
(i) there exists $j > i$ such that $\mathsf{match}(e, T(j))$; or
(ii) $\forall j > i.\ \exists k.\ k \geq j \wedge \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)$.
For (a), we know $\mathsf{wait\text{-}free}(T)$.
For (b), for any i and e, where $e \in \mathsf{pend\_inv}(T(1..i))$, we let $\mathsf{t} = \mathsf{tid}(e)$. Since $\mathsf{iso}(T)$, we know

$$|T| \neq \omega \ \vee\ \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t})\,.$$

If $|T| \neq \omega$, we know (ii) cannot hold. Thus (i) must hold.
Otherwise, we know there exists t_0 and i_0 such that

$$\forall j.\ j \geq i_0 \implies \mathsf{tid}(T(j)) = \mathsf{t}_0\,.$$

If $\mathsf{t}_0 = \mathsf{t}$, we know (ii) does not hold, and hence (i) holds. Otherwise, if $\mathsf{t}_0 \neq \mathsf{t}$, we know

$$\forall k.\ k \geq i_0 \implies \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)\,.$$

Thus we know $\mathsf{wait\text{-}free}(T)$.

For (2) $\forall T.\ \mathsf{wait\text{-}free}(T) \implies \mathsf{obstruction\text{-}free}(T)$:

Proof. From $\mathsf{wait\text{-}free}(T)$, we know one of the following holds:
(i) there exists i such that $\mathsf{is\_abt}(T(i))$ holds; or
(ii) for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then one of the following holds:
  (1) there exists $j > i$ such that $\forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)$; or
  (2) there exists $j > i$ such that $\mathsf{match}(e, T(j))$.
For (i), we know $\mathsf{obstruction\text{-}free}(T)$ holds.
For (ii), for any i and e, where $e \in \mathsf{pend\_inv}(T(1..i))$, if (1) holds, we know

$$\forall j > i.\ \exists k.\ k \geq j \wedge \mathsf{tid}(T(k)) \neq \mathsf{tid}(e).$$

Thus we know $\mathsf{obstruction\text{-}free}(T)$.

For (3) $\forall T.\ \neg\mathsf{iso}(T) \implies \mathsf{obstruction\text{-}free}(T)$:

Proof. From $\neg\mathsf{iso}(T)$, we know

$$|T| = \omega \wedge \forall \mathsf{t}, i.\ \exists j.\ j \geq i \wedge \mathsf{tid}(T(j)) \neq \mathsf{t}\,.$$

Thus, for any i and e, where $e \in \mathsf{pend\_inv}(T(1..i))$, we know

$$\forall j.\ \exists k.\ k \geq j \wedge \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)\,.$$

Thus we have proved $\mathsf{obstruction\text{-}free}(T)$.

For (4) $\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T.\ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \wedge \mathsf{prog\text{-}s}(T) \implies \mathsf{obstruction\text{-}free}(T)$:

Proof. From $\mathsf{prog\text{-}s}(T)$, we know: for any i, if $\mathsf{pend\_inv}(T(1..i)) \neq \emptyset$, then there exists $j > i$ such that $\mathsf{is\_ret}(T(j))$.
If $|T| \neq \omega$, by Lemma 17, we know $\mathsf{obstruction\text{-}free}(T)$ hold. Otherwise, $|T| = \omega$. For any i and e such that $e \in \mathsf{pend\_inv}(T(1..i))$, we know one of the following must hold:
(1) there exists $j > i$ such that $\mathsf{match}(e, T(j))$; or
(2) $\forall j.\ j > i \Rightarrow \neg\mathsf{match}(e, T(j))$.
For (2), we know

$$\forall j.\ j > i\ \Rightarrow\ e \in \mathsf{pend\_inv}(T(1..j))\,.$$

Thus we have

$$\forall j.\ j > i\ \Rightarrow\ \exists k.\ k > j \wedge \mathsf{is\_ret}(T(k))\,.$$

Then we know

$$\forall j > i.\ \exists k.\ k > j \wedge \mathsf{is\_ret}(T(k)) \wedge \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)\,.$$

Thus we know $\mathsf{obstruction\text{-}free}(T)$.

4. The first equivalence is trivial from definition. For the second equivalence, we only need to prove the following:

$$\mathsf{non\text{-}sched}(T) \wedge \neg\mathsf{prog\text{-}s}(T) \implies \neg\mathsf{fair}(T)\,.$$

From the proof of the equivalences for wait-freedom, we know

$$(\mathsf{pend\_inv}(T) = \emptyset) \Longleftrightarrow \mathsf{prog\text{-}t}(T).$$

Thus we only need to prove the following.
(1) $\mathsf{non\text{-}sched}(T) \wedge (\mathsf{pend\_inv}(T) \neq \emptyset) \implies \neg\mathsf{fair}(T)$;
(2) $\mathsf{prog\text{-}t}(T) \implies \mathsf{prog\text{-}s}(T)$.
For (1), from the premises, we know

$$\exists e, i.\ e \in \mathsf{pend\_inv}(T) \wedge \forall j \geq i.\ \mathsf{tid}(T(j)) \neq \mathsf{tid}(e)\,.$$

Thus from the operational semantics and the generation of T, we know

$$|T| = \omega \ \wedge\ \exists \mathsf{t} \in [1..\mathsf{tnum}(T)].\ |(T|_{\mathsf{t}})| \neq \omega \wedge \mathsf{last}(T|_{\mathsf{t}}) \neq (\mathsf{t}, \mathbf{term})\,.$$

Thus $\neg\mathsf{fair}(T)$ holds.
(2) is trivial from definition.

5. The first equivalence is trivial from definition. For the second equivalence, we only need to prove the following:

$$\mathsf{non\text{-}sched}(T) \wedge \neg\mathsf{prog\text{-}t}(T) \implies \neg\mathsf{fair}(T)\,.$$

It has been proved in the proofs for the equivalences for deadlock-freedom. □

From Lemma 6, we can get most of the implications in the lattice of Figure 1. To prove the remaining implications on sequential termination, we first prove some equivalences in the sequential setting below.

Lemma 7 (Equivalences in Sequential Setting). *For any C_1, σ_c, σ_o and T, if $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1), (\sigma_c, \sigma_o, \circledcirc)]\!]$, then*

1. $\mathsf{fair}(T)$ *and* $\mathsf{iso}(T)$ *holds;*
2. $\mathsf{lock\text{-}free}(T) \Longleftrightarrow \mathsf{wait\text{-}free}(T) \Longleftrightarrow \mathsf{obstruction\text{-}free}(T) \Longleftrightarrow \mathsf{deadlock\text{-}free}(T) \Longleftrightarrow \mathsf{starvation\text{-}free}(T)$.

Proof. 1. Since $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1), (\sigma_c, \sigma_o, \circledcirc)]\!]$, by the operational semantics we know $T(1) = (\mathbf{spawn}, 1)$ and

$$\forall i.\ 2 \leq i \leq |T| \implies \mathsf{tid}(T(i)) = 1\,.$$

If $|T| = \omega$, we know $|(T|_1)| = |T| = \omega$. Thus $\mathsf{fair}(T)$ and $\mathsf{iso}(T)$.
2. By Lemma 6 and the above case. □

From Lemmas 6 and 7, we get the following theorem.

Theorem 3 (Figure 1).

1. $\mathsf{wait\text{-}free}_\varphi(\Pi) \implies \mathsf{lock\text{-}free}_\varphi(\Pi)$;
2. $\mathsf{wait\text{-}free}_\varphi(\Pi) \implies \mathsf{starvation\text{-}free}_\varphi(\Pi)$;
3. $\mathsf{lock\text{-}free}_\varphi(\Pi) \implies \mathsf{obstruction\text{-}free}_\varphi(\Pi)$;
4. $\mathsf{lock\text{-}free}_\varphi(\Pi) \implies \mathsf{deadlock\text{-}free}_\varphi(\Pi)$;
5. $\mathsf{starvation\text{-}free}_\varphi(\Pi) \implies \mathsf{deadlock\text{-}free}_\varphi(\Pi)$;
6. $\mathsf{obstruction\text{-}free}_\varphi(\Pi) \implies \mathsf{seq\text{-}term}_\varphi(\Pi)$;
7. $\mathsf{deadlock\text{-}free}_\varphi(\Pi) \implies \mathsf{seq\text{-}term}_\varphi(\Pi)$.

B.3 Proofs of Theorem ??

Lemma 8 (Finite trace must be lock-free). *For any T, if*

$$T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$$

and $|T| \neq \omega$, then $\mathsf{lock\text{-}free}(T)$ *must hold.*

Proof. Suppose $T = (\textbf{spawn}, n) :: T'$. We know one of the following holds:

(i) $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T'}{\longmapsto}{}^* \textbf{abort}$; or
(ii) $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T'}{\longmapsto}{}^* (\textbf{skip}, \_)$.

For either case, we can prove $\mathsf{lock\text{-}free}(T)$ by the operational semantics. □

We define the MGC version of lock-freedom.

Definition 7. $\mathsf{lock\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$, *iff*

$$\begin{array}{l} \forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\sigma_o \in dom(\varphi)) \\ \Longrightarrow\ (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))) \end{array}$$

We use $\mathsf{get\_objevt}(T)$ to project T to the sub-trace of object events (including method invocation, return, object fault, and normal object actions). Thus we know:

$$\forall T, T'.\ (\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')) \implies (\mathsf{get\_hist}(T) = \mathsf{get\_hist}(T'))\,.$$

The following lemma is similar to Lemma 1 (MGC is the most general). But here we take into account infinite traces generated by complete executions.

Lemma 9. *For any T, if*

$$T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

then one of the following holds:

(1) $|T| \neq \omega$; or
(2) there exists i such that $\forall j \geq i$. $\mathsf{is\_clt}(T(j))$*; or*
(3) there exists T_m such that

$$T_m \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]\,,$$

and $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$.

Proof. By co-induction over $T \in \mathcal{T}_\omega[\![W, \mathcal{S}]\!]$, where

$$(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\_, \_, \circledcirc)) \overset{-}{\longmapsto}{}^* (W, \mathcal{S}) \wedge (W \neq \textbf{skip})\,.$$

In other words, $(W, \mathcal{S})$ is a "well-formed" configuration. We only need to prove the following (B.7):

$$\begin{array}{l}\text{for any } T,\ W,\ \mathcal{S},\ W_m \text{ and } \mathcal{S}_m, \text{ if}\\ \\ \text{(a) } (W,\mathcal{S}) \precsim_{\mathsf{MGC}} (W_m,\mathcal{S}_m),\\ \text{(b) } (W,\mathcal{S}) \overset{T}{\longmapsto}{}^{\omega} \cdot, \text{ and}\\ \text{(c) } \forall i.\ \exists j.\ j \geq i \wedge \neg \mathsf{is\_clt}(T(j)) \wedge T(j) \neq (\_, \mathbf{term}),\\ \\ \text{then there exists } T_m \text{ such that } (W_m,\mathcal{S}_m) \overset{T_m}{\longmapsto}{}^{\omega} \cdot \text{ and } \mathsf{get\_objevt}(T) =\\ \mathsf{get\_objevt}(T_m).\end{array} \tag{B.7}$$

Here $\precsim_{\mathsf{MGC}}$ is defined in Figure 11(a). We first prove $\precsim_{\mathsf{MGC}}$ is a simulation:

$$\begin{array}{l}\text{If } (W,\mathcal{S}) \precsim_{\mathsf{MGC}} (W_m,\mathcal{S}_m) \text{ and } (W,\mathcal{S}) \overset{e}{\longmapsto} (W',\mathcal{S}'), \text{ then}\\ \text{there exist } T, W'_m, \mathcal{S}'_m \text{ such that } (W_m,\mathcal{S}_m) \overset{T}{\longmapsto}{}^{*} (W'_m,\mathcal{S}'_m)\,,\\ \mathsf{get\_objevt}(e) = \mathsf{get\_objevt}(T) \text{ and}\\ (W',\mathcal{S}') \precsim_{\mathsf{MGC}} (W'_m,\mathcal{S}'_m)\,.\end{array} \tag{B.8}$$

This is proved by case analysis of e.

- If $e = (\mathsf{t}, \mathbf{out}, n)$ or $e = (\mathsf{t}, \mathbf{clt})$ or $e = (\mathsf{t}, \mathbf{term})$, we know the call stack of the current thread t (which makes the step) is $\circ$, before and after the step. Then we simply let $(W_m, \mathcal{S}_m)$ go zero step, and hence $T = \epsilon$. Thus $\mathsf{get\_objevt}(e) = \mathsf{get\_objevt}(T)$ and we can prove $(W',\mathcal{S}') \precsim_{\mathsf{MGC}} (W_m,\mathcal{S}_m)$.
- If $e = (\mathsf{t}, f_i, n)$, we know the call stack of the thread t is $\circ$ before the step and is (σ_l, x, C') after the step. Then we know the code of t in W_m must be MGT. We let it go two steps. After the first step, the code of t becomes $f_{\mathbf{rand}(m)}(\mathbf{rand}())$; MGT. We evaluate $\mathbf{rand}(m)$ to i and $\mathbf{rand}()$ to n, and make the second step. Thus the resulting configuration satisfies $(W',\mathcal{S}') \precsim_{\mathsf{MGC}} (W'_m,\mathcal{S}'_m)$, and $T = e$.
- If $e = (\mathsf{t}, \mathbf{ret}, n)$, we know the call stack of the thread t is (σ_l, x, C') before the step and is $\circ$ after the step. Then we let the code of t in W_m go two steps. After the first step, the code of t becomes $\mathbf{skip}$; MGT. After the second step, we have $(W',\mathcal{S}') \precsim_{\mathsf{MGC}} (W'_m,\mathcal{S}'_m)$. Also we know the first step generates the event e, and thus $\mathsf{get\_objevt}(e) = \mathsf{get\_objevt}(T)$.
- If $e = (\mathsf{t}, \mathbf{obj})$, we know the call stack of the thread t is not $\circ$ before or after the step. We let the code of t in W_m go one step, and hence $T = (\mathsf{t}, \mathbf{obj})$ and $(W',\mathcal{S}') \precsim_{\mathsf{MGC}} (W'_m,\mathcal{S}'_m)$.

Thus we have proved (B.8).

From (B.8), we can prove the following by induction over the steps of T:

$$\begin{array}{l}\text{If } (W,\mathcal{S}) \precsim_{\mathsf{MGC}} (W_m,\mathcal{S}_m)\,, (W,\mathcal{S}) \overset{T}{\longmapsto}{}^{+} (W',\mathcal{S}') \text{ and}\\ (\exists i.\ \neg \mathsf{is\_clt}(T(i)) \wedge T(i) \neq (\_, \mathbf{term}))\,, \text{ then}\\ \text{there exist } T_m, W'_m, \mathcal{S}'_m \text{ such that } (W_m,\mathcal{S}_m) \overset{T_m}{\longmapsto}{}^{+} (W'_m,\mathcal{S}'_m)\,,\\ \mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m) \text{ and } (W',\mathcal{S}') \precsim_{\mathsf{MGC}} (W'_m,\mathcal{S}'_m)\,.\end{array}$$

Then we can get (B.7) by co-induction.

When $(W, \mathcal{S}) = (\lfloor \textbf{let } \Pi \textbf{ in } C_1 \parallel \ldots \parallel C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc))$, we know $(W, \mathcal{S}) \precsim_{\mathsf{MGC}} (\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc))$. Thus we are done. □

We prove that the MGC version is equivalent to the original version of lock-freedom.

Lemma 10. $\mathsf{lock\text{-}free}_\varphi(\Pi) \iff \mathsf{lock\text{-}free}_\varphi^{\mathsf{MGC}}(\Pi)$.

Proof. 1. $\mathsf{lock\text{-}free}_\varphi(\Pi) \implies \mathsf{lock\text{-}free}_\varphi^{\mathsf{MGC}}(\Pi)$:

We prove the following:

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\sigma_o \in dom(\varphi)) \wedge \mathsf{lock\text{-}free}(T) \\ \implies (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))\end{array} \tag{B.9}$$

We unfold $\mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, then we have three cases:

(1) $(\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \_$

(2) $(\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc)) \xmapsto{T}{}^{*} (\textbf{skip}, \_)$

(3) $(\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc)) \xmapsto{T}{}^{*} \textbf{abort}$

We know from the operational semantics that (2) is impossible.

For (3), we know from the operational semantics that $\mathsf{last}(T) = (\_, \textbf{obj}, \textbf{abort})$. Thus $\exists i.\ \mathsf{is\_obj\_abt}(T(i))$.

For (1), we prove the following by contradiction:

$$\begin{array}{l}\forall n, \sigma_o, T.\ (\lfloor \textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n \rfloor, (\emptyset, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \_ \\ \implies \forall i.\ \exists j.\ j \geq i \wedge (\mathsf{is\_inv}(T(j)) \vee \mathsf{is\_ret}(T(j)) \vee T(j) = (\_, \textbf{obj}))\end{array} \tag{B.10}$$

Then, $\forall i.\ \exists j.\ j \geq i \wedge (\mathsf{is\_ret}(T(j)) \vee \mathsf{pend\_inv}(T(1..j)) \neq \emptyset)$. Thus by $\mathsf{lock\text{-}free}(T)$, we are done.

2. $\mathsf{lock\text{-}free}_\varphi^{\mathsf{MGC}}(\Pi) \implies \mathsf{lock\text{-}free}_\varphi(\Pi)$:

For any $T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, by Lemma 9, we know one of the following holds:

(1) $|T| \neq \omega$; or

(2) there exists i such that $\forall j \geq i.\ \mathsf{is\_clt}(T(j))$; or

(3) there exists T_m such that

$$T_m \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!],$$

and $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$.

For (1), by Lemma 8, we know $\mathsf{lock\text{-}free}(T)$.

For (2), we know $\mathsf{lock\text{-}free}(T)$ holds immediately by definition.

For (3), from $\mathsf{lock\text{-}free}_\varphi^{\mathsf{MGC}}(\Pi)$, we know

$$(\exists i.\ \mathsf{is\_obj\_abt}(T_m(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T_m(j))).$$

Thus we have:

$$(\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))).$$

If $\exists i.\ \mathsf{is\_obj\_abt}(T(i))$, we know $\mathsf{lock\text{-}free}(T)$. Otherwise, we know

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)).$$

Thus, for any i, if $\mathsf{pend\_inv}(T(1..i)) \neq \emptyset$, then there exists $j > i$ such that $\mathsf{is\_ret}(T(j))$. Therefore $\mathsf{lock\text{-}free}(T)$ and we are done.

□

Then, we only need to prove the following (B.11), (B.12) and (B.13):

$$\Pi \sqsubseteq^{\omega}_{\varphi} \Pi_A \implies \Pi \sqsubseteq_{\varphi} \Pi_A \tag{B.11}$$

$$\Pi \sqsubseteq^{\omega}_{\varphi} \Pi_A \implies \mathsf{lock\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi) \tag{B.12}$$

$$\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{lock\text{-}free}_{\varphi}(\Pi) \implies \Pi \sqsubseteq^{\omega}_{\varphi} \Pi_A \tag{B.13}$$

Proofs of (B.11) For any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, for any T if

$$T \in \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

we know there exists T_1 such that $T = \mathsf{get\_obsv}(T_1)$ and

$$T_1 \in \mathcal{T}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

Thus there exists T_1' and T_1'' such that $T_1'' = T_1 :: T_1'$ and one of the following holds:

(i) $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{\omega} \cdot$; or
(ii) $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{*} (\mathbf{skip}, \_)$; or
(iii) $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{*} \mathbf{abort}$.

That is,

$$T_1'' \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

Since $\Pi \sqsubseteq^{\omega}_{\varphi} \Pi_A$, we know there exists T_2'' such that

$$T_2'' \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$

and

$$\mathsf{get\_obsv}(T_2'') = \mathsf{get\_obsv}(T_1'') = T :: \mathsf{get\_obsv}(T_1')\,.$$

Thus there exists T_2 such that

$$T_2 \in \mathcal{T}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$

and $\mathsf{get\_obsv}(T_2) = T$. Thus

$$T \in \mathcal{O}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$

and we are done.

Proofs of (B.12) We construct another most general client as follows:

$$\mathsf{MGTp1} \stackrel{\text{def}}{=} \textbf{while}\ (\textbf{true})\{\ f_{\textbf{rand}(m)}(\textbf{rand}());\ \textbf{print}(1);\ \}$$
$$\mathsf{MGCp1}_n \stackrel{\text{def}}{=} \|_{i\in[1..n]}\,\mathsf{MGTp1}$$

The following lemma describes the relationship between MGCp1 and MGC:

Lemma 11. *(1) For any T, if*

$$T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!],$$

then there exists T_p such that

$$T_p \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!],$$

$T_p\backslash(\_, \textbf{out}, 1) = T$ *and*

$$\forall i, \mathsf{t}.\ T_p(i) = (\mathsf{t}, \textbf{ret}, \_) \ \Leftrightarrow\ T_p(i+1) = (\mathsf{t}, \textbf{out}, 1)\,.$$

(2) For any T_p, if

$$T_p \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!],$$

then there exists T such that

$$T \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi\ \textbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$$

and $T_p\backslash(\_, \textbf{out}, 1) = T$.

Here we use $T_p\backslash(\_, \textbf{out}, 1)$ to mean a sub-trace of T_p which removes all the events of the form $(\_, \textbf{out}, 1)$.

Proof. By constructing simulations between executions of **let** Π **in** MGC_n and **let** Π **in** $\mathsf{MGCp1}_n$. □

Lemma 12. *Suppose Π_A is total.*
For any n, σ_a and T, if $T \in \mathcal{O}_\omega[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$, then T is an infinite trace of $(\_, \textbf{out}, 1)$.

Proof. We need to prove: for any T such that
$T \in \mathcal{O}_\omega[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$, the following hold:

(1) $|T| = \omega$;
(2) for any i, $T(i) = (\_, \textbf{out}, 1)$.

For (1): we can prove for any T' such that

$$T' \in \mathcal{T}_\omega[\![(\textbf{let}\ \Pi_A\ \textbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!],$$

we have $|T'| = \omega$. If $|T| \neq \omega$, we know there exists i such that

$$\forall j \geq i.\ \mathsf{is\_inv}(T'(j)) \vee \mathsf{is\_ret}(T'(j)) \vee T'(j) = (\_, \textbf{obj}) \vee T'(j) = (\_, \textbf{clt}).$$

Since Π_A is total, from the code and the operational semantics, we know this is impossible.

(2) is easily proved from $|T| = \omega$ and that the code can only produce $(\_, \mathbf{out}, 1)$ as observable events. □

To prove $\mathsf{lock\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$, we want to show: for any n, σ_o, σ_a and T, if $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\varphi(\sigma_o) = \sigma_a$, then

$$(\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))) \tag{B.14}$$

First, if $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, by Lemma 11(1), there exists T_p such that $T_p \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $T_p \backslash (\_, \mathbf{out}, 1) = T$.

Since $\Pi \sqsubseteq^{\omega}_{\varphi} \Pi_A$, we know

$$\mathcal{O}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

From Lemma 12, we know for any T, if $T \in \mathcal{O}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, then T is an infinite trace of $(\_, \mathbf{out}, 1)$.

Then we know: $\mathsf{get\_obsv}(T_p)$ is an infinite trace of $(\_, \mathbf{out}, 1)$.
Thus $|T_p| = \omega$ and

$$\forall i.\ \exists j.\ j \geq i \wedge T_p(j) = (\_, \mathbf{out}, 1)\,. \tag{B.15}$$

We prove the following:

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T_p(j))\,. \tag{B.16}$$

This is proved as follows. From $|T_p| = \omega$ and (B.15), we know for any i, there exist $j_1, \ldots, j_{n+1}$ such that $i \leq j_1 < \ldots < j_{n+1}$ and $\forall k \in [1..n+1].\ T_p(j_k) = (\_, \mathbf{out}, 1)$. Then, by the pigeonhole principle, we know there exists a thread t producing two $(\mathsf{t}, \mathbf{out}, 1)$-s. Suppose j_k and j_l are the indexes of the two events produced by t and $j_k < j_l$. By the operational semantics, we know there exists j' such that $i \leq j_k < j' < j_l$ and $\mathsf{is\_ret}(T_p(j'))$. Thus we have proved (B.16).

Since $T_p \backslash (\_, \mathbf{out}, 1) = T$, from (B.16), we know (B.14) holds and we are done.

Proofs of (B.13) We need to prove that if $\Pi \sqsubseteq_\varphi \Pi_A$ and $\mathsf{lock\text{-}free}_\varphi(\Pi)$, then for any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, we have

$$\begin{array}{l}\mathcal{O}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

Thus we only need to prove: for any T,

(1) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \mathbf{abort}$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \mathbf{abort}$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(2) If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\textbf{skip}, \_)$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\textbf{skip}, \_)$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(3) If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

Actually neither (1) or (2) depends on progress properties. We can prove the following lemma.

Lemma 13. *If $\Pi \sqsubseteq_\varphi \Pi_A$, then for any n, C_1, ..., C_n, σ_c, σ_o, σ_a and T such that $\varphi(\sigma_o) = \sigma_a$, we have*

1. *If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \textbf{abort}$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \textbf{abort}$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.*
2. *If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\textbf{skip}, \_)$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\textbf{skip}, \_)$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.*

Proof. 1. We know $\mathsf{is\_abt}(\mathsf{last}(T))$. By $\Pi \sqsubseteq_\varphi \Pi_A$, we know there exists T_a such that

$$T_a \in \mathcal{T}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]$$

and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$. Thus $\mathsf{is\_abt}(\mathsf{last}(T_a))$, and by the operational semantics, we know

$$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \textbf{abort},$$

and we are done.

2. (a) If $n = 1$, we know

$$(\textbf{let } \Pi \textbf{ in } \{C; \textbf{end}\}, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\textbf{skip}, \_).$$

Thus there exists T'' such that $T = T'' :: (1, \textbf{term})$. Let

$$T' = T'' :: (1, \textbf{clt}) :: (1, \textbf{out}, \text{“done”}) :: (1, \textbf{clt}) :: (1, \textbf{term}),$$

where we assume $(1, \textbf{out}, \text{“done”})$ is different from all the events in T, then

$$(\textbf{let } \Pi \textbf{ in } \{C; \textbf{print}(\text{“done”}); \textbf{end}\}, (\sigma_c, \sigma_o, \circledcirc)) \overset{T'}{\longmapsto}{}^* (\textbf{skip}, \_).$$

Since $\Pi \sqsubseteq_\varphi \Pi_A$, we know there exists T'_a such that

$$T'_a \in \mathcal{T}[\![(\textbf{let } \Pi_A \textbf{ in } \{C; \textbf{print}(\text{“done”}); \textbf{end}\}), (\sigma_c, \sigma_a, \circledcirc)]\!]$$

and $\mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T'_a)$. Thus we know there exists T''_a such that

$$T'_a = T''_a :: (1, \textbf{out}, \text{“done”}) :: (1, \textbf{clt}) :: (1, \textbf{term}),$$

and by the operational semantics, we know there exists T_a such that $T_a'' = T_a :: (1, \mathbf{clt})$ and

$$(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \{C; \mathbf{end}\}, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a :: (1, \mathbf{term})}{\longmapsto}{}^* (\mathbf{skip}, \_).$$

Also we have $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(b) If $n > 1$, we construct another program $\mathbf{let}\ \Pi\ \mathbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n'$ as follows: we pick $n-1$ fresh variables: $\mathtt{d}_2, \ldots, \mathtt{d}_n$,

$$\begin{aligned} C_1' &= (C_1; \mathbf{if}\ (\mathtt{d}_2 \&\& \ldots \&\& \mathtt{d}_n)\ \mathbf{print}(\text{“done”});) \\ C_i' &= (C_i; \mathtt{d}_i := \mathbf{true}) \qquad \forall i \in [2..n] \end{aligned}$$

and also let

$$\sigma_c' = \sigma_c \uplus \{\mathtt{d}_2 \rightsquigarrow \mathbf{false}, \ldots, \mathtt{d}_n \rightsquigarrow \mathbf{false}\}.$$

Then, if

$$(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\mathbf{skip}, \_),$$

let T'' be the result after removing all the termination markers in T, and

$$\begin{aligned} T' = T'' &:: (2, \mathbf{clt}) :: (2, \mathbf{clt}) :: \ldots :: (n, \mathbf{clt}) :: (n, \mathbf{clt}) \\ &:: (1, \mathbf{clt}) :: (1, \mathbf{clt}) :: (1, \mathbf{out}, \text{“done”}) \\ &:: (1, \mathbf{clt}) :: (1, \mathbf{term}) :: \ldots :: (n, \mathbf{clt}) :: (n, \mathbf{term}) \end{aligned}$$

where we still assume $(1, \mathbf{out}, \text{“done”})$ is different from all the events in T, we can prove:

$$(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n' \rfloor, (\sigma_c', \sigma_o, \circledcirc)) \overset{T'}{\longmapsto}{}^* (\mathbf{skip}, \_).$$

Since $\Pi \sqsubseteq_\varphi \Pi_A$, we know there exists T_a' such that

$$T_a' \in \mathcal{T}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n'), (\sigma_c', \sigma_a, \circledcirc)]\!]$$

and $\mathsf{get\_obsv}(T') = \mathsf{get\_obsv}(T_a')$. Thus we know there exists i such that $T_a'(i) = (1, \mathbf{out}, \text{“done”})$. Then we know

$$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1' \,\|\, \ldots \,\|\, C_n' \rfloor, (\sigma_c', \sigma_a, \circledcirc)) \overset{T_a'}{\longmapsto}{}^* (\mathbf{skip}, \_).$$

We can remove all the actions of the newly added commands, construct a simulation between the two executions, and prove: there exists T_a such that

$$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\mathbf{skip}, \_),$$

and $\mathsf{get\_obsv}(T_a) = \mathsf{get\_obsv}(T_a'') = \mathsf{get\_obsv}(T)$.

Thus we are done. □

For (3), we define the simulation relation $\precsim$ in Figure 11(d), and prove the following (B.17) by case analysis and the operational semantics:

$$\begin{aligned} &\text{For any } W_1,\ \mathcal{S}_1,\ W_2,\ \mathcal{S}_2,\ W_3,\ \mathcal{S}_3 \text{ and } e_1, \\ &\text{if } (W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3) \text{ and } (W_1, \mathcal{S}_1) \overset{e_1}{\longmapsto} (W_1', \mathcal{S}_1'), \\ &\text{then there exist } T_2,\ W_2',\ \mathcal{S}_2',\ T_3,\ W_3' \text{ and } \mathcal{S}_3' \text{ such that} \\ &(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^* (W_2', \mathcal{S}_2'),\ (W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^* (W_3', \mathcal{S}_3'), \\ &T_3 \backslash (\_, \mathbf{obj}) = e_1 \backslash (\_, \mathbf{obj}) \text{ and } (W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3'). \end{aligned} \tag{B.17}$$

With (B.17), we can prove the following (B.18) by induction over the length of T_1:

For any W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$ and T_1,
if $(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^{+} (W_1', \mathcal{S}_1')$ and
$\mathsf{last}(T_1) \neq (\_, \mathbf{obj})$,
then there exist T_2, W_2', $\mathcal{S}_2'$, T_3, W_3' and $\mathcal{S}_3'$ such that
$(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^{*} (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{+} (W_3', \mathcal{S}_3')$,
$T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj})$ and $(W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$.
(B.18)

With (B.18), we can prove the following (B.19):

For any W, $\mathcal{S}$, W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$, T_0 and T_1,
if $(W, \mathcal{S})$ is well-formed and out of method calls, $(W, \mathcal{S}) \overset{T_0}{\longmapsto}{}^{*} (W_1, \mathcal{S}_1)$,
$(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{lock\text{-}free}(T_0 :: T_1)$,
then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{\omega} \cdot$ and
$T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj})$.
(B.19)

We prove (B.19) as follows. Let $T = T_0 :: T_1$. Since $\mathsf{lock\text{-}free}(T)$, we know one of the following holds:

(i) there exists i such that $\forall j \geq i.\ \mathsf{is\_clt}(T(j))$; or
(ii) for any i, if $\mathsf{pend\_inv}(T(1..i)) \neq \emptyset$, then there exists $j > i$ such that $\mathsf{is\_ret}(T(j))$.

For (i), we know there exist W_1', $\mathcal{S}_1'$, T_1' and T_1'' such that

$$\begin{gathered}(W_1, \mathcal{S}_1) \overset{T_1'}{\longmapsto}{}^{+} (W_1', \mathcal{S}_1'), \quad (W_1', \mathcal{S}_1') \overset{T_1''}{\longmapsto}{}^{\omega} \cdot, \\ T_1 = T_1' :: T_1'', \quad T_1' = T_1(1..i), \quad \mathsf{is\_clt}(\mathsf{last}(T_1')), \quad \forall j.\ \mathsf{is\_clt}(T_1''(j)).\end{gathered}$$

By (B.18), we know: there exist T_2, W_2', $\mathcal{S}_2'$, T_3', W_3' and $\mathcal{S}_3'$ such that $(W_2, \mathcal{S}_2) \overset{T_2}{\longmapsto}{}^{*} (W_2', \mathcal{S}_2')$, $(W_3, \mathcal{S}_3) \overset{T_3'}{\longmapsto}{}^{+} (W_3', \mathcal{S}_3')$, $T_1' \backslash (\_, \mathbf{obj}) = T_3' \backslash (\_, \mathbf{obj})$ and $(W_1', \mathcal{S}_1') \precsim (W_2', \mathcal{S}_2'; W_3', \mathcal{S}_3')$. Then by coinduction over T_1 and from (B.18), we get: there exists T_3'' such that

$$(W_3', \mathcal{S}_3') \overset{T_3''}{\longmapsto}{}^{\omega} \cdot \text{ and } T_1'' \backslash (\_, \mathbf{obj}) = T_3'' \backslash (\_, \mathbf{obj}).$$

Let $T_3 = T_3' :: T_3''$, and we know

$$(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{\omega} \cdot \text{ and } T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj}).$$

Suppose (i) does not hold. Thus we know

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_obj}(T(j)).$$

By the operational semantics, we know

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{pend\_inv}(T(1..j)) \neq \emptyset.$$

Since (ii) holds, we know

$$\forall i.\ \exists j.\ j > i \wedge \mathsf{is\_ret}(T(j)).$$

Then by coinduction and from (B.18), we know there exists T_3 such that

$$(W_3, \mathcal{S}_3) \xmapsto{T_3}{}^{\omega} \cdot \text{ and } T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj}).$$

Thus we have proved (B.19). On the other hand, for any n, C_1, ..., C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] .$$

From $\Pi \sqsubseteq_\varphi \Pi_A$, by Lemma 3, we know $\Pi \subseteqq_\varphi \Pi_A$. Thus, if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!] .$$

Then we know

$$\begin{array}{l}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc); \\ \quad \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)),\end{array}$$

Thus, if $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T}{}^{\omega} \cdot$, by $\mathsf{lock\text{-}free}_\varphi(\Pi)$, we know $\mathsf{lock\text{-}free}(T)$. Then from (B.19) we get: there exists T_a such that

$$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{T_a}{}^{\omega} \cdot$$

and $T \backslash (\_, \mathbf{obj}) = T_a \backslash (\_, \mathbf{obj})$. Thus $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and we are done.

B.4 Proofs of Theorem ??

Similar to Lemma 8, we can prove the following lemma.

Lemma 14 (Finite trace must be wait-free). *For any T, if*

$$T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$$

and $|T| \neq \omega$, then $\mathsf{wait\text{-}free}(T)$ *must hold.*

We define the MGC version of wait-freedom, and prove it is equivalent to the original version.

Definition 8. $\mathsf{wait\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi)$*, iff*

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\sigma_o \in \mathit{dom}(\varphi)) \\ \Longrightarrow \mathsf{wait\text{-}free}(T)\end{array}$$

Lemma 15. $\mathsf{wait\text{-}free}_\varphi(\Pi) \iff \mathsf{wait\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi)$.

Proof. 1. $\mathsf{wait\text{-}free}_\varphi(\Pi) \implies \mathsf{wait\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi)$:
Trivial.

2. $\mathsf{wait\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi) \implies \mathsf{wait\text{-}free}_\varphi(\Pi)$:
For any $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$, by Lemma 9, we know one of the following holds:

(1) $|T| \neq \omega$; or
(2) there exists i such that $\forall j \geq i.\ \mathsf{is\_clt}(T(j))$; or
(3) there exists T_m such that
$$T_m \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]\,,$$
and $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$.

For (1), by Lemma 14, we know $\mathsf{wait\text{-}free}(T)$ holds.

For (2), we know $|T| = \omega$.

For any k and e, if $e \in \mathsf{pend\_inv}(T(1..k))$, we know one of the following must hold:
(i) $\exists j.\ j > k \wedge \mathsf{match}(e, T(j))$.
(ii) $\forall j.\ j > k \Rightarrow \neg\mathsf{match}(e, T(j))$. Thus we can prove:
$$\forall j \geq k.\ e \in \mathsf{pend\_inv}(T(1..j)).$$
Let $l = \mathsf{max}(i, k)$. Then we know:
$$\forall j \geq l.\ \mathsf{is\_clt}(T(j)) \wedge e \in \mathsf{pend\_inv}(T(1..j)).$$
Thus by the operational semantics, we can prove:
$$\forall j > l.\ \mathsf{tid}(T(j)) \neq \mathsf{tid}(e).$$
Thus we know $\mathsf{wait\text{-}free}(T)$.

For (3), suppose (1) does not hold for T, and we only need to prove the following:
for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then there exists $j > i$ such that either $\forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{tid}(e)$ or $\mathsf{match}(e, T(j))$.

From $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, we know
$$\neg\exists i.\ \mathsf{is\_obj\_abt}(T_m(i)).$$
Then by the operational semantics and the generation of T_m, we know
$$\neg\exists i.\ \mathsf{is\_abt}(T_m(i)).$$
From $\mathsf{wait\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$, we know $\mathsf{wait\text{-}free}(T_m)$, then we have
for any i and e, if $e \in \mathsf{pend\_inv}(T_m(1..i))$, then there exists $j > i$ such that either $\forall k \geq j.\ \mathsf{tid}(T_m(k)) \neq \mathsf{tid}(e)$ or $\mathsf{match}(e, T_m(j))$.

For any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, we know there exists i_m such that
$$e \in \mathsf{pend\_inv}(T_m(1..i_m)) \text{ and } \mathsf{get\_objevt}(T(1..i)) = \mathsf{get\_objevt}(T_m(1..i_m)).$$
We know there exists $j_m > i_m$ such that one of the following holds:
(i) $\mathsf{match}(e, T_m(j_m))$; or
(ii) $\forall k \geq j_m.\ \mathsf{tid}(T_m(k)) \neq \mathsf{tid}(e)$.

For (i), since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, we know there exists $j > i$ such that $\mathsf{match}(e, T(j))$.

For (ii), suppose
$$\forall j > i.\ \neg\mathsf{match}(e, T(j)) \text{ and } \forall j > i.\ \exists k \geq j.\ \mathsf{tid}(T(k)) = \mathsf{tid}(e)\,.$$
Since $e \in \mathsf{pend\_inv}(T(1..i))$, by the operational semantics, we know
$$\forall j > i.\ \exists k \geq j.\ T(k) = (\mathsf{tid}(e), \mathbf{obj})\,.$$

Since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, we know

$$\forall j > i_m.\ \exists k \geq j.\ T_m(k) = (\mathsf{tid}(e), \mathbf{obj})\,,$$

which contradicts (ii). Thus we get $\mathsf{wait\text{-}free}(T)$ and we are done.

□

Then, we only need to prove the following (B.20), (B.21) and (B.22):

$$\mathit{\Pi} \sqsubseteq_{\varphi}^{t\omega} \mathit{\Pi}_A \implies \mathit{\Pi} \sqsubseteq_{\varphi}^{\omega} \mathit{\Pi}_A \tag{B.20}$$

$$\mathit{\Pi} \sqsubseteq_{\varphi}^{t\omega} \mathit{\Pi}_A \implies \mathsf{wait\text{-}free}_{\varphi}^{\mathsf{MGC}}(\mathit{\Pi}) \tag{B.21}$$

$$\mathit{\Pi} \sqsubseteq_{\varphi} \mathit{\Pi}_A \wedge \mathsf{wait\text{-}free}_{\varphi}(\mathit{\Pi}) \implies \mathit{\Pi} \sqsubseteq_{\varphi}^{t\omega} \mathit{\Pi}_A \tag{B.22}$$

Proofs of (B.20) For any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, for any T, suppose

$$T \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

Since $\mathit{\Pi} \sqsubseteq_{\varphi}^{t\omega} \mathit{\Pi}_A$, we know there exists T_a such that

$$T_a \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$
$$\mathsf{get\_obsv}(T_a) = \mathsf{get\_obsv}(T) \text{ and } \mathsf{div\_tids}(T_a) = \mathsf{div\_tids}(T)\,.$$

Thus we are done.

Proofs of (B.21) Just like the proofs of (B.12), we use the most general client $\mathsf{MGCp1}$. We first prove the following lemma:

Lemma 16. *Suppose $\mathit{\Pi}_A$ is total.*
For any n, σ_a, T and S, if $(T, S) \in \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]$, then $\mathsf{div\_tids}(T) = S$.

Proof. We know there exists T_1 such that

$$T_1 \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!],$$
$$T = \mathsf{get\_obsv}(T_1) \text{ and } S = \mathsf{div\_tids}(T_1).$$

It's easy to see that $\mathsf{div\_tids}(T) \subseteq S$.
On the other hand, for all $\mathsf{t} \in S$, we know:

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{tid}(T_1(j)) = \mathsf{t}\,.$$

By the operational semantics and the generation of T_1, we know

$$\forall i.\ \exists j.\ j \geq i \wedge T_1(j) = (\mathsf{t}, \mathbf{out}, 1)\,.$$

Thus we can prove:

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{tid}(T(j)) = \mathsf{t}\,.$$

Thus $\mathsf{t} \in \mathsf{div\_tids}(T)$, and we are done. □

For any n, σ_o, σ_a and T such that $\varphi(\sigma_o) = \sigma_a$, if

$$T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!],$$

by Lemma 11(1), there exists T_p such that

$$T_p \in \mathcal{T}_\omega[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \text{ and } T_p \backslash (\_, \mathbf{out}, 1) = T.$$

Suppose $\neg\exists i.\ \mathsf{is\_abt}(T(i))$.
Then for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, we know there exists i_p such that

$$e \in \mathsf{pend\_inv}(T_p(1..i_p)) \text{ and } (T_p(1..i_p)) \backslash (\_, \mathbf{out}, 1) = T(1..i).$$

Let $\mathsf{t} = \mathsf{tid}(e)$, we suppose

$$\forall j > i.\ \exists k \geq j.\ \mathsf{tid}(T(k)) = \mathsf{tid}(e) = \mathsf{t}\,.$$

Since $T_p \backslash (\_, \mathbf{out}, 1) = T$, we know:

$$\forall j > i_p.\ \exists k \geq j.\ \mathsf{tid}(T_p(k)) = \mathsf{t}\,.$$

Thus we know

$$\mathsf{t} \in \mathsf{div\_tids}(T_p)\,.$$

On the other hand, since $\mathit{\Pi} \sqsubseteq^{t\omega}_{\varphi} \mathit{\Pi}_A$, we know:

$$\mathcal{O}_{t\omega}[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

Then from Lemma 16, we know

$$\mathsf{div\_tids}(T_p) = \mathsf{div\_tids}(\mathsf{get\_obsv}(T_p)).$$

Thus

$$\mathsf{t} \in \mathsf{div\_tids}(\mathsf{get\_obsv}(T_p)),$$

and then we can prove:

$$\forall j.\ \exists k \geq j.\ T_p(k) = (\mathsf{t}, \mathbf{out}, 1)\,.$$

Then since $e \in \mathsf{pend\_inv}(T_p(1..i_p))$ and by the operational semantics, we know

$$\text{there must exist } j \text{ such that } j > i_p \text{ and } \mathsf{match}(e, T_p(j)).$$

Since $T_p \backslash (\_, \mathbf{out}, 1) = T$, we know:

$$\text{there exists } j \text{ such that } j > i \text{ and } \mathsf{match}(e, T(j)).$$

Thus $\mathsf{wait\text{-}free}(T)$ and we are done.

Proofs of (B.22) We need to prove that if $\Pi \sqsubseteq_\varphi \Pi_A$ and $\mathsf{wait\text{-}free}_\varphi(\Pi)$, then for any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, we have

$$\begin{array}{l}\mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

Thus we only need to prove: for any T,

(1) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \mathbf{abort}$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \mathbf{abort}$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(2) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\mathbf{skip}, \_)$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\mathbf{skip}, \_)$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(3) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$,
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and $\mathsf{div\_tids}(T) = \mathsf{div\_tids}(T_a)$.

(1) and (2) are proved in Lemma 13.

For (3), we define the simulation relation $\precsim$ in Figure 11(d), and as in the proof for (B.13), we can get the following (B.23) from (B.19) and the fact that $\mathsf{wait\text{-}free}(T_0 :: T_1)$ implies $\mathsf{lock\text{-}free}(T_0 :: T_1)$:

$$\begin{array}{l}\text{For any } W,\ \mathcal{S},\ W_1,\ \mathcal{S}_1,\ W_2,\ \mathcal{S}_2,\ W_3,\ \mathcal{S}_3,\ T_0 \text{ and } T_1, \\ \text{if } (W, \mathcal{S}) \text{ is well-formed and out of method calls, } (W, \mathcal{S}) \overset{T_0}{\longmapsto}{}^* (W_1, \mathcal{S}_1), \\ (W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3),\ (W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^\omega \cdot \text{ and } \mathsf{wait\text{-}free}(T_0 :: T_1), \\ \text{then there exists } T_3 \text{ such that } (W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^\omega \cdot \text{ and} \\ T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj}).\end{array} \tag{B.23}$$

On the other hand, for any n, C_1, …, C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]\,.$$

From $\Pi \sqsubseteq_\varphi \Pi_A$, by Lemma 3, we know $\Pi \subseteqq_\varphi \Pi_A$. Thus, if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

Then we know

$$\begin{array}{l}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc); \\ \quad \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)),\end{array}$$

Thus, if $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$, by $\mathsf{wait\text{-}free}_\varphi(\Pi)$, we know $\mathsf{wait\text{-}free}(T)$. Then from (B.23) we get: there exists T_a such that

$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \xmapsto{T_a}{}^{\omega} \cdot$ and $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$.

Thus we know $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.
Below we prove: $\mathsf{div\_tids}(T) = \mathsf{div\_tids}(T_a)$.

(a) $\mathsf{div\_tids}(T) \subseteq \mathsf{div\_tids}(T_a)$:
For any i, since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, we know there exists i' such that $T(1..i') \backslash (\_, \textbf{obj}) = T_a(1..i) \backslash (\_, \textbf{obj})$. For any $\mathsf{t} \in \mathsf{div\_tids}(T)$, we know

$$\exists j'.\ j' \geq i' \wedge \mathsf{tid}(T(j')) = \mathsf{t}\,.$$

If $T(j') \neq (\mathsf{t}, \textbf{obj})$, since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, we know there exists $j \geq i$ such that $T_a(j) = T(j')$.
Otherwise, $T(j') = (\mathsf{t}, \textbf{obj})$. By the operational semantics and the generation of T, we know there exists e such that

$$e \in \mathsf{pend\_inv}(T(1..j'-1)) \text{ and } \mathsf{tid}(e) = \mathsf{t}\,.$$

Since $\mathsf{wait\text{-}free}(T)$, we know one of the following holds:
(i) there exists $l \geq j'$ such that $\forall k \geq l.\ \mathsf{tid}(T(k)) \neq \mathsf{t}$; or
(ii) there exists $j'' \geq j'$ such that $\mathsf{match}(e, T(j''))$.
Suppose (i) holds. Since $\mathsf{t} \in \mathsf{div\_tids}(T)$, we know

$$\exists j''.\ j'' \geq l \wedge \mathsf{tid}(T(j'')) = \mathsf{t}\,,$$

which is a contradiction.
Thus (ii) must hold. Thus $T(j'') = (\mathsf{t}, \textbf{ret}, \_)$ and $j'' \geq i'$. Since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, we know there exists $j \geq i$ such that $T_a(j) = T(j'')$.
Thus we have proved

$$\exists j.\ j \geq i \wedge \mathsf{tid}(T_a(j)) = \mathsf{t}\,.$$

Therefore $\mathsf{t} \in \mathsf{div\_tids}(T_a)$.

(b) $\mathsf{div\_tids}(T_a) \subseteq \mathsf{div\_tids}(T_a)$:
For any i', since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, we know there exists i such that $T(1..i') \backslash (\_, \textbf{obj}) = T_a(1..i) \backslash (\_, \textbf{obj})$. For any $\mathsf{t} \in \mathsf{div\_tids}(T_a)$, we know

$$\exists j.\ j \geq i \wedge \mathsf{tid}(T_a(j)) = \mathsf{t}\,.$$

If $T_a(j) \neq (\mathsf{t}, \textbf{obj})$, since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, we know there exists $j' \geq i'$ such that $T_a(j) = T(j')$.
Otherwise, $T_a(j) = (\mathsf{t}, \textbf{obj})$. By the operational semantics and the generation of T_a, we know one of the following holds:
(i) $\forall k > j.\ \mathsf{tid}(T_a(k)) \neq \mathsf{t}$; or
(ii) there exists $j'' \geq j$ such that $\mathsf{match}(e, T_a(j''))$.
Suppose (i) holds. Since $\mathsf{t} \in \mathsf{div\_tids}(T_a)$, we know

$$\exists j''.\ j'' > j \wedge \mathsf{tid}(T_a(j'')) = \mathsf{t}\,,$$

which is a contradiction.
Thus (ii) must hold. Thus $T_a(j'') = (\mathsf{t}, \textbf{ret}, \_)$ and $j'' \geq i$. Since $T \backslash (\_, \textbf{obj}) = T_a \backslash (\_, \textbf{obj})$, we know there exists $j' \geq i'$ such that $T_a(j'') = T(j')$.
Thus we have proved

$$\exists j'.\ j' \geq i' \wedge \mathsf{tid}(T(j')) = \mathsf{t}\,.$$

Therefore $\mathsf{t} \in \mathsf{div\_tids}(T)$.

Thus we are done.

B.5 Proofs of Theorem ??

Lemma 17 (Finite trace must be obstruction-free). *For any T, if*

$$T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]$$

and $|T| \neq \omega$, then $\mathsf{obstruction\text{-}free}(T)$ *must hold.*

We define the MGC version of obstruction-freedom, and prove it is equivalent to the original version.

Definition 9. $\mathsf{obstruction\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$*, iff*

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge \mathsf{iso}(T) \wedge (\sigma_o \in dom(\varphi)) \\ \implies (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))\,.\end{array}$$

Lemma 18. $\mathsf{obstruction\text{-}free}_{\varphi}(\Pi) \iff \mathsf{obstruction\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$.

Proof. From Figure 7, we know $\mathsf{obstruction\text{-}free}_{\varphi}(\Pi)$ is equivalent to the following:

$$\begin{array}{l}\forall n, C_1, \ldots, C_n, \sigma_c, \sigma_o, T. \\ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o)]\!] \wedge \mathsf{iso}(T) \wedge (\sigma_o \in dom(\varphi)) \\ \implies \mathsf{lock\text{-}free}(T)\end{array}$$

By Lemma 10, we know it is equivalent to the following:

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge \mathsf{iso}(T) \wedge (\sigma_o \in dom(\varphi)) \\ \implies (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))\,.\end{array}$$

Thus we are done. □

Then, we only need to prove the following (B.24), (B.25) and (B.26):

$$\Pi \sqsubseteq^{i\omega}_{\varphi} \Pi_A \implies \Pi \sqsubseteq_{\varphi} \Pi_A \tag{B.24}$$

$$\Pi \sqsubseteq^{i\omega}_{\varphi} \Pi_A \implies \mathsf{obstruction\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi) \tag{B.25}$$

$$\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{obstruction\text{-}free}_{\varphi}(\Pi) \implies \Pi \sqsubseteq^{i\omega}_{\varphi} \Pi_A \tag{B.26}$$

Proofs of (B.24) For any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, for any T if

$$T \in \mathcal{O}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

we know there exists T_1 such that $T = \mathsf{get\_obsv}(T_1)$ and

$$T_1 \in \mathcal{T}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

Thus there exists T_1' and T_1'' such that $T_1'' = T_1 :: T_1'$, where $\mathsf{iso}(T_1')$ holds, and one of the following holds:

(i) $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{\omega} \cdot$; or

(ii) $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T_1''}{\longmapsto}{}^* (\textbf{skip}, \_)$; or
(iii) $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T_1''}{\longmapsto}{}^* \textbf{abort}$.

Thus,

$$T_1'' \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \| \ldots \| C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \text{ and } \mathsf{iso}(T_1'') \,.$$

Since $\Pi \sqsubseteq_\varphi^{i\omega} \Pi_A$, we know there exists T_2'' such that

$$T_2'' \in \mathcal{T}_\omega[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n), (\sigma_c, \sigma_a, \circledcirc)]\!] \,,$$

and

$$\mathsf{get\_obsv}(T_2'') = \mathsf{get\_obsv}(T_1'') = T :: \mathsf{get\_obsv}(T_1') \,.$$

Thus there exists T_2 such that

$$T_2 \in \mathcal{T}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n), (\sigma_c, \sigma_a, \circledcirc)]\!] \,,$$

and $\mathsf{get\_obsv}(T_2) = T$. Thus

$$T \in \mathcal{O}[\![(\textbf{let } \Pi_A \textbf{ in } C_1 \| \ldots \| C_n), (\sigma_c, \sigma_a, \circledcirc)]\!] \,,$$

and we are done.

Proofs of (B.25) The proof is similar to the proof of (B.12).

To prove $\mathsf{obstruction\text{-}free}_\varphi^{\mathsf{MGC}}(\Pi)$, we want to show: for any n, σ_o, σ_a and T, if $T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, $\mathsf{iso}(T)$ and $\varphi(\sigma_o) = \sigma_a$, then the following (B.14) holds:

$$(\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))) \,.$$

First, if $T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\mathsf{iso}(T)$, by Lemma 11(1), there exists T_p such that

$$T_p \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!], \quad T_p \backslash (\_, \textbf{out}, 1) = T$$
$$\text{and} \quad \forall i, \mathsf{t}.\ T_p(i) = (\mathsf{t}, \textbf{ret}, \_) \ \Leftrightarrow\ T_p(i+1) = (\mathsf{t}, \textbf{out}, 1).$$

Since $\mathsf{iso}(T)$, we know

$$|T| = \omega \implies \exists \mathsf{t}, i.\ (\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t}) \,.$$

If $|T_p| = \omega$, by the generation of T_p and $T_p \backslash (\_, \textbf{out}, 1) = T$, we know $|T| = \omega$. Thus there exist t_0 and i such that

$$\forall j.\ j \geq i \implies \mathsf{tid}(T(j)) = \mathsf{t}_0 \,.$$

Since $T_p \backslash (\_, \textbf{out}, 1) = T$, we know there exists i_p such that

$$\forall j.\ j \geq i_p \implies \mathsf{tid}(T_p(j)) = \mathsf{t}_0 \vee T_p(j) = (\_, \textbf{out}, 1) \,.$$

By the generation of T_p, we know there exists i' such that

$$\forall j.\ j \geq i' \implies \mathsf{tid}(T_p(j)) = \mathsf{t}_0\,.$$

Thus $\mathsf{iso}(T_p)$ holds.

Since $\Pi \sqsubseteq_\varphi^{i\omega} \Pi_A$, we know

$$\mathcal{O}_{i\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_{\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

From Lemma 12, we know for any T, if $T \in \mathcal{O}_{i\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, then T is an infinite trace of $(\_, \mathbf{out}, 1)$.

Then we know: $\mathsf{get\_obsv}(T_p)$ is an infinite trace of $(\_, \mathbf{out}, 1)$. Thus $|T_p| = \omega$ and the following (B.15) holds:

$$\forall i.\ \exists j.\ j \geq i \wedge T_p(j) = (\_, \mathbf{out}, 1)\,.$$

As in the proof of (B.12), we prove the following (B.16) from (B.15):

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T_p(j))\,.$$

Since $T_p \backslash (\_, \mathbf{out}, 1) = T$, from (B.16), we know (B.14) holds and we are done.

Proofs of (B.26) We need to prove that if $\Pi \sqsubseteq_\varphi \Pi_A$ and $\mathsf{obstruction\text{-}free}_\varphi(\Pi)$, then for any n, C_1, ..., C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, we have

$$\begin{array}{l}\mathcal{O}_{i\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}_{\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

Thus we only need to prove: for any T,

(1) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \mathbf{abort}$, then there exists T_a such that $(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \mathbf{abort}$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(2) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\mathbf{skip}, \_)$, then there exists T_a such that $(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\mathbf{skip}, \_)$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(3) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$ and $\mathsf{iso}(T)$, then there exists T_a such that $(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(1) and (2) are proved in Lemma 13.

For (3), as in the proofs for (B.13), we define the simulation relation $\precsim$ in Figure 11(d), and prove the following (B.19):

For any W, $\mathcal{S}$, W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$, T_0 and T_1,
if $(W, \mathcal{S})$ is well-formed and out of method calls, $(W, \mathcal{S}) \overset{T_0}{\longmapsto}{}^* (W_1, \mathcal{S}_1)$,
$(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^\omega \cdot$ and $\mathsf{lock\text{-}free}(T_0 :: T_1)$,
then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^\omega \cdot$ and
$T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj})$.

On the other hand, for any n, C_1, ..., C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] .$$

From $\Pi \sqsubseteq_\varphi \Pi_A$, by Lemma 3, we know $\Pi \subseteqq_\varphi \Pi_A$. Thus, if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!] .$$

Then we know

$$\begin{array}{l}(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc); \\ \quad \textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)),\end{array}$$

Thus, if $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{iso}(T)$,
by $\mathsf{obstruction\text{-}free}_\varphi(\Pi)$, we know $\mathsf{lock\text{-}free}(T)$. Then from (B.19) we get: there exists T_a such that

$$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} .$$

and $T\backslash(\_, \textbf{obj}) = T_a\backslash(\_, \textbf{obj})$. Thus $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and we are done.

B.6 Proofs of Theorem ??

We define the MGC version of deadlock-freedom, and prove it is equivalent to the original version.

Definition 10. $\mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi)$, *iff*

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge \mathsf{objfair}(T) \wedge (\sigma_o \in \mathit{dom}(\varphi)) \\ \Longrightarrow\ (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))) ,\end{array}$$

where $\mathsf{objfair}(T)$ *says object steps are fairly scheduled:*

$$\begin{array}{rcl}\mathsf{objfair}(T) & \overset{\text{def}}{=} & |T| = \omega \\ & & \Longrightarrow\ (\forall \mathsf{t} \in [1..\mathsf{tnum}(T)].\ \forall n.\ |(T|_{\mathsf{t}})| = n \\ & & \quad \Longrightarrow\ \mathsf{is\_ret}((T|_{\mathsf{t}})(n)) \vee \mathsf{is\_clt}((T|_{\mathsf{t}})(n)) \vee (T|_{\mathsf{t}})(n) = (\mathsf{t}, \textbf{term})) .\end{array}$$

It's easy to see:

$$\forall T.\ \mathsf{fair}(T) \implies \mathsf{objfair}(T) .$$

Lemma 19. *For any* T *and* T_m, *if* $\mathsf{fair}(T)$, $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, $|T| = \omega$ *and*

$$T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] ,$$

then $\mathsf{objfair}(T_m)$.

Proof. Suppose $|(T_m|_{\mathsf{t}})| = n$ and the index of $(T_m|_{\mathsf{t}})(n)$ in T_m is l. If $\mathsf{is\_ret}(T_m(l))$ or $\mathsf{is\_clt}(T_m(l))$ or $T_m(l) = (\mathsf{t}, \textbf{term})$, we are done. Otherwise, we know

$$\mathsf{is\_inv}(T_m(l)) \text{ or } T_m(l) = (\mathsf{t}, \textbf{obj}) .$$

Since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, we know there exists i such that

$$T(i) = T_m(l) \quad \text{and} \quad \mathsf{get\_objevt}(T(1..i)) = \mathsf{get\_objevt}(T_m(1..l))\,.$$

Thus $\mathsf{tid}(T(i)) = \mathsf{t}$ and

$$\mathsf{is\_inv}(T(i)) \text{ or } T(i) = (\mathsf{t}, \mathbf{obj})\,.$$

From $\mathsf{fair}(T)$, we know

$$\exists j.\ j > i \wedge \mathsf{tid}(T(j)) = \mathsf{t}\,.$$

By the generation of T and the operational semantics, we know

$$\exists j.\ j > i \wedge \mathsf{tid}(T(j)) = \mathsf{t} \wedge \mathsf{is\_obj}(T(j))\,.$$

Since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$, we know

$$\exists j.\ j > l \wedge \mathsf{tid}(T_m(j)) = \mathsf{t} \wedge \mathsf{is\_obj}(T_m(j))\,,$$

which contradicts the assumption that $|(T_m|_\mathsf{t})| = n$ and the index of $(T_m|_\mathsf{t})(n)$ in T_m is l. Thus neither $\mathsf{is\_inv}(T_m(l))$ nor $T_m(l) = (\mathsf{t}, \mathbf{obj})$ holds, and we are done. □

Lemma 20. $\mathsf{deadlock\text{-}free}_\varphi(\Pi) \iff \mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi)$.

Proof. 1. $\mathsf{deadlock\text{-}free}_\varphi(\Pi) \implies \mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi)$:
As in the proof for Lemma 10, we can prove the following (B.9):

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge (\sigma_o \in dom(\varphi)) \wedge \mathsf{lock\text{-}free}(T) \\ \implies (\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))\end{array}$$

Then we only need to prove the following (B.27):

$$\begin{array}{l}\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \\ \wedge\, \mathsf{objfair}(T) \wedge (\sigma_o \in dom(\varphi)) \wedge \mathsf{deadlock\text{-}free}_\varphi(\Pi) \\ \implies \mathsf{lock\text{-}free}(T)\end{array} \tag{B.27}$$

For T such that $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\mathsf{objfair}(T)$, if $|T| \neq \omega$, then we know $\mathsf{fair}(T)$. By the definition of $\mathsf{deadlock\text{-}free}_\varphi(\Pi)$, we know $\mathsf{lock\text{-}free}(T)$. Otherwise, we know $|T| = \omega$, and let

$$\begin{array}{rcl} S & \stackrel{\text{def}}{=} & \{\mathsf{t} \mid \exists n.\ |(T|_\mathsf{t})| = n \wedge (T|_\mathsf{t})(n) \neq (\mathsf{t}, \mathbf{term})\} \\ & = & \{\mathsf{t} \mid |(T|_\mathsf{t})| \neq \omega\}\,. \end{array}$$

Then we construct another program $W = \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n$ as follows: for any $\mathsf{t} \in [1..n]$,

$$\begin{array}{l} \mathsf{t} \notin S \ \Rightarrow\ C_\mathsf{t} = \mathsf{MGT} \\ \mathsf{t} \in S \\ \Rightarrow\ C_\mathsf{t} = \ \text{local } i_\mathsf{t};\ i_\mathsf{t} := 0; \\ \qquad\quad \mathbf{while}\ (i_\mathsf{t} < n_\mathsf{t})\{\ f_{\mathbf{rand}(m)}(\mathbf{rand}()); i_\mathsf{t} := i_\mathsf{t} + 1\ \} \\ \qquad\qquad\qquad\qquad \text{where } n_\mathsf{t} = |\mathsf{get\_hist}(T|_\mathsf{t})|/2 \end{array}$$

Let $\sigma_c = \{i_{\mathsf{t}} \rightsquigarrow 0 \mid \mathsf{t} \in S\}$.
We can construct a simulation between **let** Π **in** MGC_n and W, and show that there exists T' such that

$$T' \in \mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o, \circledcirc)]\!], \quad \mathsf{fair}(T') \quad \text{and} \quad \mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T').$$

From $\mathsf{deadlock\text{-}free}_\varphi(\Pi)$, we know $\mathsf{lock\text{-}free}(T')$. We can prove the following (B.28):

$$\begin{array}{l}\text{If } |T| = \omega,\ \mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T') \text{ and } \mathsf{lock\text{-}free}(T'), \\ \text{then } \mathsf{lock\text{-}free}(T).\end{array} \tag{B.28}$$

Then we know $\mathsf{lock\text{-}free}(T)$ and hence (B.27) holds.
We prove (B.28) as follows. Since $|T| = \omega$, we know one of the following must hold:
(i) there exists i such that $\forall j \geq i.\ \mathsf{is\_clt}(T(j))$;
(ii) $\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_obj}(T(j))$.
For (i), we know $\mathsf{lock\text{-}free}(T)$.
For (ii), since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')$, we know

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_obj}(T'(j)).$$

Since $\mathsf{lock\text{-}free}(T')$, we know
for any i', if $\mathsf{pend\_inv}(T'(1..i')) \neq \emptyset$, then there exists $j' > i'$ such that $\mathsf{is\_ret}(T'(j'))$.
For T, for any i, we know there exists i' such that

$$\mathsf{get\_objevt}(T(1..i)) = \mathsf{get\_objevt}(T'(1..i')).$$

If $\mathsf{pend\_inv}(T(1..i)) \neq \emptyset$, we know

$$\mathsf{pend\_inv}(T'(1..i')) \neq \emptyset.$$

Then we get:

$$\text{there exists } j' > i' \text{ such that } \mathsf{is\_ret}(T'(j')).$$

Thus we know:

$$\text{there exists } j > i \text{ such that } \mathsf{is\_ret}(T(j)).$$

Therefore $\mathsf{lock\text{-}free}(T)$ and we have proved (B.28).

2. $\mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_\varphi(\Pi) \implies \mathsf{deadlock\text{-}free}_\varphi(\Pi)$:
For any T such that

$$T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

by Lemma 9, we know one of the following holds:
(1) $|T| \neq \omega$; or
(2) there exists i such that $\forall j \geq i.\ \mathsf{is\_clt}(T(j))$; or
(3) there exists T_m such that

$$T_m \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!],$$

and $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$.
For (1), by Lemma 8, we know $\mathsf{lock\text{-}free}(T)$.
For (2), we know $\mathsf{lock\text{-}free}(T)$ holds immediately by definition.
For (3), suppose (1) does not hold. If $\mathsf{fair}(T)$, by Lemma 19, we know $\mathsf{objfair}(T_m)$ holds. Then from $\mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$, we know

$$(\exists i.\ \mathsf{is\_obj\_abt}(T_m(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T_m(j))).$$

Thus we have:

$$(\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))).$$

If $\exists i.\ \mathsf{is\_obj\_abt}(T(i))$, we know $\mathsf{lock\text{-}free}(T)$. Otherwise, we know

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)).$$

Thus, for any i, if $\mathsf{pend\_inv}(T(1..i)) \neq \emptyset$, then there exists $j > i$ such that $\mathsf{is\_ret}(T(j))$. Therefore $\mathsf{lock\text{-}free}(T)$ and we are done.

□

Then, we only need to prove the following (B.29), (B.30) and (B.31):

$$\Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A \implies \Pi \sqsubseteq_{\varphi} \Pi_A \tag{B.29}$$

$$\Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A \implies \mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi) \tag{B.30}$$

$$\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{deadlock\text{-}free}_{\varphi}(\Pi) \implies \Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A \tag{B.31}$$

Proofs of (B.29) For any n, C_1, ..., C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, for any T if

$$T \in \mathcal{O}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

we know there exists T_1 such that $T = \mathsf{get\_obsv}(T_1)$ and

$$T_1 \in \mathcal{T}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,.$$

Thus there exists T_1' and T_1'' such that $T_1'' = T_1 :: T_1'$, where $\mathsf{fair}(T_1')$ holds, and one of the following holds:

(i) $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{\omega} \cdot$; or
(ii) $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{*} (\mathbf{skip}, \_)$; or
(iii) $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \xmapsto{T_1''}{}^{*} \mathbf{abort}$.

Thus,

$$T_1'' \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \quad \text{and} \quad \mathsf{fair}(T_1'')\,.$$

Since $\Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A$, we know there exists T_2'' such that

$$T_2'' \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$

and

$$\mathsf{get\_obsv}(T_2'') = \mathsf{get\_obsv}(T_1'') = T :: \mathsf{get\_obsv}(T_1')\,.$$

Thus there exists T_2 such that

$$T_2 \in \mathcal{T}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$

and $\mathsf{get\_obsv}(T_2) = T$. Thus

$$T \in \mathcal{O}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,,$$

and we are done.

Proofs of (B.30) The proof is similar to the proof of (B.12), except that we need to first prove the following lemma:

Lemma 21. *Suppose $\mathit{\Pi}_A$ is total. If $\mathit{\Pi} \sqsubseteq^{f\omega}_{\varphi} \mathit{\Pi}_A$, then*

$$\mathcal{O}_{of\omega}[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_{\omega}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,,$$

where

$$\mathcal{O}_{of\omega}[\![W, \mathcal{S}]\!] \stackrel{\text{def}}{=} \{\mathsf{get\_obsv}(T) \mid T \in \mathcal{T}_{\omega}[\![W, \mathcal{S}]\!] \wedge \mathsf{objfair}(T) \wedge \forall i, \mathsf{t}.\ T(i) = (\mathsf{t}, \mathbf{ret}, \_) \Leftrightarrow T(i+1) = (\mathsf{t}, \mathbf{out}, 1)\}\,.$$

Proof. For any T and T_o such that

$$T \in \mathcal{T}_{\omega}[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!], \quad \mathsf{objfair}(T),$$
$$\forall i, \mathsf{t}.\ T(i) = (\mathsf{t}, \mathbf{ret}, \_) \Leftrightarrow T(i+1) = (\mathsf{t}, \mathbf{out}, 1),$$

and $T_o = \mathsf{get\_obsv}(T)$, if $|T| \neq \omega$, we know $\mathsf{fair}(T)$ holds, thus

$$T_o \in \mathcal{O}_{f\omega}[\![(\mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!].$$

From $\mathit{\Pi} \sqsubseteq^{f\omega}_{\varphi} \mathit{\Pi}_A$, we know

$$T_o \in \mathcal{O}_{\omega}[\![(\mathbf{let}\ \mathit{\Pi}_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!].$$

Otherwise, we know $|T| = \omega$, and let

$$\begin{aligned} S &\stackrel{\text{def}}{=} \{\mathsf{t} \mid \exists n.\ |(T|_{\mathsf{t}})| = n \wedge (T|_{\mathsf{t}})(n) \neq (\mathsf{t}, \mathbf{term})\} \\ &= \{\mathsf{t} \mid |(T|_{\mathsf{t}})| \neq \omega\}\,. \end{aligned}$$

Since $|T| = \omega$, we know there exists t such that $|(T|_{\mathsf{t}})| = \omega$ and hence $\mathsf{t} \notin S$. Then we construct another program $W = \mathbf{let}\ \mathit{\Pi}\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n$ as follows: for any $\mathsf{t} \in [1..n]$,

$$\begin{array}{l} \mathsf{t} \notin S \ \Rightarrow\ C_{\mathsf{t}} = \mathsf{MGTp1} \\ \mathsf{t} \in S \\ \Rightarrow\ C_{\mathsf{t}} = \ \mathrm{local}\ i_{\mathsf{t}};\ i_{\mathsf{t}} := 0; \\ \qquad\qquad \mathbf{while}\ (i_{\mathsf{t}} < n_{\mathsf{t}})\{ \\ \qquad\qquad\quad f_{\mathbf{rand}(m)}(\mathbf{rand}()); \mathbf{print}(1); i_{\mathsf{t}} := i_{\mathsf{t}} + 1; \\ \qquad\qquad \} \\ \qquad\qquad\qquad\qquad\qquad \text{where } n_{\mathsf{t}} = |\mathsf{get\_hist}(T|_{\mathsf{t}})|/2 \end{array}$$

Let $\sigma_c = \{i_{\mathsf{t}} \rightsquigarrow 0 \mid \mathsf{t} \in S\}$.
We can construct a simulation between $\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n$ and W, and show that there exists T' such that

$$T' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,,$$
$$\mathsf{fair}(T') \quad \text{and} \quad \mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T') = T_o\,.$$

Since $\Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A$, we know

$$T_o \in \mathcal{O}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_a, \circledcirc)]\!].$$

Thus there exists T'' such that

$$T'' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \parallel \ldots \parallel C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,, \quad \text{and} \quad \mathsf{get\_obsv}(T'') = T_o\,.$$

Since there exists t such that $C_{\mathsf{t}} = \mathsf{MGCp1}$, we can construct a simulation and show that there exists T''' such that

$$T''' \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,,$$
$$\text{and} \quad \mathsf{get\_obsv}(T'') = \mathsf{get\_obsv}(T''') = T_o\,.$$

Thus we are done. □

To prove $\mathsf{deadlock\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi)$, we want to show: for any n, σ_o, σ_a and T, if $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, $\mathsf{objfair}(T)$ and $\varphi(\sigma_o) = \sigma_a$, then the following (B.14) holds:

$$(\exists i.\ \mathsf{is\_obj\_abt}(T(i))) \vee (\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j)))\,. \tag{B.14}$$

First, if $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\mathsf{objfair}(T)$, by Lemma 11(1), there exists T_p such that

$$T_p \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!], \quad T_p \backslash (\_, \mathbf{out}, 1) = T$$
$$\text{and} \quad \forall i, \mathsf{t}.\ T_p(i) = (\mathsf{t}, \mathbf{ret}, \_) \ \Leftrightarrow\ T_p(i+1) = (\mathsf{t}, \mathbf{out}, 1).$$

Since $\mathsf{objfair}(T)$, we know $\mathsf{objfair}(T_p)$ also holds.

Since $\Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A$, by Lemma 21, we know

$$\mathcal{O}_{of\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

From Lemma 12, we know for any T, if $T \in \mathcal{O}_{of\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$, then T is an infinite trace of $(\_, \mathbf{out}, 1)$.

Then we know: $\mathsf{get\_obsv}(T_p)$ is an infinite trace of $(\_, \mathbf{out}, 1)$. Thus $|T_p| = \omega$ and the following (B.15) holds:

$$\forall i.\ \exists j.\ j \geq i \wedge T_p(j) = (\_, \mathbf{out}, 1)\,. \tag{B.15}$$

As in the proof of (B.12), we prove the following (B.16) from (B.15):

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T_p(j))\,. \tag{B.16}$$

Since $T_p \backslash (\_, \mathbf{out}, 1) = T$, from (B.16), we get (B.14) and thus we are done.

Proofs of (B.31) We need to prove that if $\Pi \sqsubseteq_\varphi \Pi_A$ and $\mathsf{deadlock\text{-}free}_\varphi(\Pi)$, then for any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, we have

$$\begin{array}{l}\mathcal{O}_{f\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}_{\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

Thus we only need to prove: for any T,

(1) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \mathbf{abort}$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \mathbf{abort}$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(2) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\mathbf{skip}, \_)$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\mathbf{skip}, \_)$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(3) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$ and $\mathsf{fair}(T)$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(1) and (2) are proved in Lemma 13.

For (3), as in the proofs for (B.13), we define the simulation relation $\precsim$ in Figure 11(d), and prove the following (B.19):

> For any W, $\mathcal{S}$, W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$, T_0 and T_1,
> if $(W, \mathcal{S})$ is well-formed and out of method calls, $(W, \mathcal{S}) \overset{T_0}{\longmapsto}{}^* (W_1, \mathcal{S}_1)$,
> $(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^\omega \cdot$ and $\mathsf{lock\text{-}free}(T_0 :: T_1)$,
> then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^\omega \cdot$ and
> $T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj})$.

On the other hand, for any n, C_1, …, C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]\,.$$

From $\Pi \sqsubseteq_\varphi \Pi_A$, by Lemma 3, we know $\Pi \subseteq_\varphi \Pi_A$. Thus, if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

Then we know

$$\begin{array}{l}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc); \\ \quad \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)),\end{array}$$

Thus, if $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$ and $\mathsf{fair}(T)$,
by $\mathsf{deadlock\text{-}free}_\varphi(\Pi)$, we know $\mathsf{lock\text{-}free}(T)$. Then from (B.19) we get: there exists T_a such that

$$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega\,.$$

and $T \backslash (\_, \mathbf{obj}) = T_a \backslash (\_, \mathbf{obj})$. Thus $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and we are done.

B.7 Proofs of Theorem ??

We define the MGC version of starvation-freedom, and prove it is equivalent to the original version.

Definition 11. $\mathsf{starvation\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$, *iff*

$$\forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \wedge \mathsf{objfair}(T) \wedge (\sigma_o \in dom(\varphi)) \\ \Longrightarrow \mathsf{wait\text{-}free}(T)$$

Lemma 22. $\mathsf{starvation\text{-}free}_{\varphi}(\Pi) \iff \mathsf{starvation\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$.

Proof. 1. $\mathsf{starvation\text{-}free}_{\varphi}(\Pi) \implies \mathsf{starvation\text{-}free}_{\varphi}^{\mathsf{MGC}}(\Pi)$:
We only need to prove the following (B.32):

$$\begin{array}{l} \forall n, \sigma_o, T.\ \ T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \\ \wedge\, \mathsf{objfair}(T) \wedge (\sigma_o \in dom(\varphi)) \wedge \mathsf{starvation\text{-}free}_{\varphi}(\Pi) \\ \Longrightarrow \mathsf{wait\text{-}free}(T) \end{array} \tag{B.32}$$

For T such that $T \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\mathsf{objfair}(T)$, if $|T| \neq \omega$, then we know $\mathsf{fair}(T)$. By the definition of $\mathsf{starvation\text{-}free}_{\varphi}(\Pi)$, we know $\mathsf{wait\text{-}free}(T)$. Otherwise, we know $|T| = \omega$, and let

$$\begin{array}{rcl} S & \stackrel{\text{def}}{=} & \{\mathsf{t} \mid \exists n.\ |(T|_{\mathsf{t}})| = n \wedge (T|_{\mathsf{t}})(n) \neq (\mathsf{t}, \mathbf{term})\} \\ & = & \{\mathsf{t} \mid |(T|_{\mathsf{t}})| \neq \omega\}\,. \end{array}$$

Then we construct another program $W = \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n$ as follows: for any $\mathsf{t} \in [1..n]$,

$$\begin{array}{l} \mathsf{t} \notin S \ \Rightarrow\ C_{\mathsf{t}} = \mathsf{MGT} \\ \mathsf{t} \in S \\ \Rightarrow\ C_{\mathsf{t}} = \ \mathrm{local}\ i_{\mathsf{t}};\ i_{\mathsf{t}} := 0; \\ \qquad\qquad \mathbf{while}\ (i_{\mathsf{t}} < n_{\mathsf{t}})\{\ f_{\mathbf{rand}(m)}(\mathbf{rand}()); i_{\mathsf{t}} := i_{\mathsf{t}} + 1\ \} \\ \qquad\qquad\qquad\qquad \text{where } n_{\mathsf{t}} = |\mathsf{get\_hist}(T|_{\mathsf{t}})|/2 \end{array}$$

Let $\sigma_c = \{i_{\mathsf{t}} \rightsquigarrow 0 \mid \mathsf{t} \in S\}$.
We can construct a simulation between $\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n$ and W, and show that there exists T' such that

$$T' \in \mathcal{T}_\omega[\![W, (\sigma_c, \sigma_o, \circledcirc)]\!]\,, \quad \mathsf{fair}(T') \quad \text{and} \quad \mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')\,.$$

From $\mathsf{starvation\text{-}free}_{\varphi}(\Pi)$, we know $\mathsf{wait\text{-}free}(T')$. We can prove the following (B.33):

$$\begin{array}{l} \text{If } |T| = \omega,\ \mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T') \text{ and } \mathsf{wait\text{-}free}(T')\,, \\ \text{then } \mathsf{wait\text{-}free}(T)\,. \end{array} \tag{B.33}$$

Then we know $\mathsf{wait\text{-}free}(T)$ and hence (B.32) holds.
We prove (B.33) as follows. Since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')$, for any i, we know there exists i' such that

$$\mathsf{get\_objevt}(T(1..i)) = \mathsf{get\_objevt}(T'(1..i'))\,.$$

For any e, if $e \in \mathsf{pend\_inv}(T(1..i))$, we know

$$e \in \mathsf{pend\_inv}(T'(1..i'))\,.$$

From $\mathsf{wait\text{-}free}(T')$, we know one of the following holds:
(i) there exists $j' > i'$ such that $\mathsf{match}(e, T'(j'))$.
(ii) there exists $j' > i'$ such that $\forall k' \geq j'.\ \mathsf{tid}(T'(k')) \neq \mathsf{tid}(e)$.
For (i), since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')$, we know

$$\text{there exists } j > i \text{ such that } \mathsf{match}(e, T(j)).$$

For (ii), assume (i) does not hold. Then we know $e \in \mathsf{pend\_inv}(T')$. Since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')$, we can prove

$$e \in \mathsf{pend\_inv}(T)\,.$$

Let $\mathsf{t} = \mathsf{tid}(e)$. Suppose

$$\forall j > i.\ \exists k \geq j.\ \mathsf{tid}(T(k)) = \mathsf{t}\,.$$

Then, by the operational semantics and the generation of T, we know

$$\forall j > i.\ \exists k \geq j.\ T(k) = (\mathsf{t}, \mathbf{obj})\,.$$

Since $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T')$, we know

$$\forall j' > i'.\ \exists k' \geq j'.\ T'(k') = (\mathsf{t}, \mathbf{obj})\,,$$

which contradicts (ii). Thus we know

$$\exists j > i.\ \forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{t}\,.$$

Therefore $\mathsf{wait\text{-}free}(T)$ and we have proved (B.33).

2. $\mathsf{starvation\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi) \implies \mathsf{starvation\text{-}free}_{\varphi}(\Pi)$:
Almost the same as the proof for Lemma 15, except that we need to apply Lemma 19.

□

Then, we only need to prove the following (B.34), (B.35) and (B.36), where (B.34) is trivial from definitions:

$$\Pi \sqsubseteq^{ft\omega}_{\varphi} \Pi_A \implies \Pi \sqsubseteq^{f\omega}_{\varphi} \Pi_A \tag{B.34}$$

$$\Pi \sqsubseteq^{ft\omega}_{\varphi} \Pi_A \implies \mathsf{starvation\text{-}free}^{\mathsf{MGC}}_{\varphi}(\Pi) \tag{B.35}$$

$$\Pi \sqsubseteq_{\varphi} \Pi_A \wedge \mathsf{starvation\text{-}free}_{\varphi}(\Pi) \implies \Pi \sqsubseteq^{ft\omega}_{\varphi} \Pi_A \tag{B.36}$$

Proofs of (B.35) For any n, σ_o, σ_a and T such that $\varphi(\sigma_o) = \sigma_a$, if $T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!]$ and $\mathsf{objfair}(T)$, suppose

$$\neg\exists i.\ \mathsf{is\_abt}(T(i)),$$

then by the operational semantics, we only need to prove:

> for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then there exists $j > i$ such that $\mathsf{match}(e, T(j))$.

Suppose it does not hold. Then we know there exists t_0 such that

$$\exists i.\ \forall j.\ j \geq i \Rightarrow (T|_{\mathsf{t}_0})(j) = (\mathsf{t}_0, \textbf{obj})\,.$$

By Lemma 11(1), there exists T_p such that

$$T_p \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n), (\emptyset, \sigma_o, \circledcirc)]\!]\,, \quad T_p\backslash(\_, \textbf{out}, 1) = T$$
$$\text{and} \quad \forall i, \mathsf{t}.\ T_p(i) = (\mathsf{t}, \textbf{ret}, \_) \ \Leftrightarrow\ T_p(i+1) = (\mathsf{t}, \textbf{out}, 1).$$

By the operational semantics, we know

$$\exists i.\ \forall j.\ j \geq i \Rightarrow (T_p|_{\mathsf{t}_0})(j) = (\mathsf{t}_0, \textbf{obj})\,.$$

Let

$$\begin{array}{rcl} S & \stackrel{\text{def}}{=} & \{\mathsf{t} \mid \exists n.\ |(T_p|_{\mathsf{t}})| = n \wedge (T_p|_{\mathsf{t}})(n) \neq (\mathsf{t}, \textbf{term})\} \\ & = & \{\mathsf{t} \mid |(T_p|_{\mathsf{t}})| \neq \omega\}\,. \end{array}$$

Thus we know

$$\mathsf{t}_0 \notin S, \ \text{ and } \mathsf{t}_0 \in \mathsf{div\_tids}(T_p).$$

We construct another program $W = \textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n$ as follows: for any $\mathsf{t} \in [1..n]$,

$$\begin{array}{l} \mathsf{t} \notin S \ \Rightarrow\ C_{\mathsf{t}} = \mathsf{MGTp1} \\ \mathsf{t} \in S \\ \Rightarrow\ C_{\mathsf{t}} = \ \text{local } i_{\mathsf{t}};\ i_{\mathsf{t}} := 0; \\ \qquad\quad \textbf{while } (i_{\mathsf{t}} < n_{\mathsf{t}})\{ \\ \qquad\quad\ f_{\textbf{rand}(m)}(\textbf{rand}()); \textbf{print}(1); i_{\mathsf{t}} := i_{\mathsf{t}} + 1; \\ \qquad\quad \} \\ \qquad\qquad\qquad\qquad \text{where } n_{\mathsf{t}} = |\mathsf{get\_hist}(T|_{\mathsf{t}})|/2 \end{array}$$

Let $\sigma_c = \{i_{\mathsf{t}} \rightsquigarrow 0 \mid \mathsf{t} \in S\}$. We can construct a simulation between $\textbf{let } \Pi \textbf{ in } \mathsf{MGCp1}_n$ and W, and show that there exists T_p' such that

$$T_p' \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!]\,, \quad \mathsf{fair}(T_p')\,,$$
$$\mathsf{get\_objevt}(T_p) = \mathsf{get\_objevt}(T_p') \quad \text{and} \quad \mathsf{get\_obsv}(T_p) = \mathsf{get\_obsv}(T_p')\,.$$

Thus we know there exists i such that

$$\forall j.\ j \geq i \Rightarrow (T_p'|_{\mathsf{t}_0})(j) = (\mathsf{t}_0, \textbf{obj})\,.$$

Thus we have

$$\mathsf{t}_0 \in \mathsf{div\_tids}(T'_p) \;\text{ and }\; |(\mathsf{get\_obsv}(T'_p)|_{\mathsf{t}_0})| < i\,.$$

On the other hand, since $\Pi \sqsubseteq_\varphi^{ft\omega} \Pi_A$, we know:

$$\begin{array}{l}\mathcal{O}_{ft\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

Thus there exists T''_p such that

$$\begin{array}{c}T''_p \in \mathcal{T}_\omega[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,, \\ \mathsf{get\_obsv}(T''_p) = \mathsf{get\_obsv}(T'_p) \quad\text{and}\quad \mathsf{div\_tids}(T''_p) = \mathsf{div\_tids}(T'_p)\,.\end{array}$$

Since $C_{\mathsf{t}_0} = \mathsf{MGTp1}$ and $\mathsf{t}_0 \in \mathsf{div\_tids}(T''_p)$, we know

$$|(T''_p|_{\mathsf{t}_0})| = \omega\,,$$

and also

$$|(\mathsf{get\_obsv}(T'_p)|_{\mathsf{t}_0})| = |(\mathsf{get\_obsv}(T''_p)|_{\mathsf{t}_0})| = \omega\,,$$

which contradicts the fact that $|(\mathsf{get\_obsv}(T'_p)|_{\mathsf{t}_0})| < i$. Thus we know $\mathsf{wait\text{-}free}(T)$ and we are done.

Proofs of (B.36) We need to prove that if $\Pi \sqsubseteq_\varphi \Pi_A$ and $\mathsf{starvation\text{-}free}_\varphi(\Pi)$, then for any n, C_1, …, C_n, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, we have

$$\begin{array}{l}\mathcal{O}_{ft\omega}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \\ \subseteq \mathcal{O}_{t\omega}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.\end{array}$$

Thus we only need to prove: for any T,

(1) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \mathbf{abort}$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \mathbf{abort}$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(2) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\mathbf{skip}, \_)$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\mathbf{skip}, \_)$ and
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(3) If $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$ and $\mathsf{fair}(T)$,
then there exists T_a such that
$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$,
$\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$ and $\mathsf{div\_tids}(T) = \mathsf{div\_tids}(T_a)$.

(1) and (2) are proved in Lemma 13.

For (3), as in the proofs for (B.22), we define the simulation relation $\precsim$ in Figure 11(d), and prove the following (B.23):

For any W, $\mathcal{S}$, W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$, T_0 and T_1,
if $(W, \mathcal{S})$ is well-formed and out of method calls, $(W, \mathcal{S}) \overset{T_0}{\longmapsto}{}^{*} (W_1, \mathcal{S}_1)$,
$(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{wait\text{-}free}(T_0 :: T_1)$,
then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^{\omega} \cdot$ and
$T_1 \backslash (\_, \mathbf{obj}) = T_3 \backslash (\_, \mathbf{obj})$.

On the other hand, for any n, C_1, ..., C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] .$$

From $\Pi \sqsubseteq_\varphi \Pi_A$, by Lemma 3, we know $\Pi \subseteq_\varphi \Pi_A$. Thus, if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\mathbf{let}\ \Pi\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n), (\emptyset, \sigma_a, \circledcirc)]\!] .$$

Then we know

$$\begin{array}{l}(\mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\mathbf{let}\ \Pi_A\ \mathbf{in}\ \mathsf{MGC}_n, (\emptyset, \sigma_a, \circledcirc); \\ \quad \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n, (\sigma_c, \sigma_a, \circledcirc)),\end{array}$$

Thus, if $(\lfloor \mathbf{let}\ \Pi\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^{\omega} \cdot$ and $\mathsf{fair}(T)$,
by $\mathsf{starvation\text{-}free}_\varphi(\Pi)$, we know $\mathsf{wait\text{-}free}(T)$. Then from (B.23) we get: there exists T_a such that

$$(\lfloor \mathbf{let}\ \Pi_A\ \mathbf{in}\ C_1 \,\|\, \ldots \,\|\, C_n \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^{\omega} \cdot, \text{ and } T \backslash (\_, \mathbf{obj}) = T_a \backslash (\_, \mathbf{obj}).$$

Thus we know $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

Below we prove: $\mathsf{div\_tids}(T) = \mathsf{div\_tids}(T_a)$. Since $\mathsf{fair}(T)$ and $|T| = \omega$, we know for any t,

$$\text{either } |(T|_{\mathsf{t}})| = \omega, \quad \text{or } \mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term}) .$$

(a) $\mathsf{last}(T|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$:
Since $T \backslash (\_, \mathbf{obj}) = T_a \backslash (\_, \mathbf{obj})$ and by the operational semantics, we know $\mathsf{last}(T_a|_{\mathsf{t}}) = (\mathsf{t}, \mathbf{term})$.

(b) $|(T|_{\mathsf{t}})| = \omega$:
Since $T \backslash (\_, \mathbf{obj}) = T_a \backslash (\_, \mathbf{obj})$, we know

$$(T|_{\mathsf{t}}) \backslash (\mathsf{t}, \mathbf{obj}) = (T_a|_{\mathsf{t}}) \backslash (\mathsf{t}, \mathbf{obj}) .$$

Suppose $|(T_a|_{\mathsf{t}})| \neq \omega$. Then we know $|(T_a|_{\mathsf{t}}) \backslash (\mathsf{t}, \mathbf{obj})| \neq \omega$. Thus

$$\exists i.\ \forall j.\ j \geq i \ \Rightarrow\ (T|_{\mathsf{t}})(j) = (\mathsf{t}, \mathbf{obj}) .$$

By the operational semantics, we know there exists i such that

$$\mathsf{tid}(T(i)) = \mathsf{t} , \quad \mathsf{is\_inv}(T(i)) , \quad \text{and} \quad \forall j.\ j \geq i \ \Rightarrow\ \neg \mathsf{match}(T(i), T(j)) .$$

By $\mathsf{wait\text{-}free}(T)$, we know

$$\exists j.\ \forall k \geq j.\ \mathsf{tid}(T(k)) \neq \mathsf{t} ,$$

which contradicts the assumption that $|(T|_{\mathsf{t}})| = \omega$.
Thus we know $|(T_a|_{\mathsf{t}})| = \omega$.

Thus $\mathsf{div\_tids}(T) = \mathsf{div\_tids}(T_a)$ holds and we are done.

B.8 Proofs of Theorem ??

Proofs of Theorem ??(1) For any σ_o, σ_a and T such that $\varphi(\sigma_o) = \sigma_a$, if

$$T \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1), (\sigma_c, \sigma_o, \circledcirc)]\!],$$

by Lemma 9, we know one of the following holds:

(1) $|T| \neq \omega$; or
(2) there exists i such that $\forall j \geq i.\ \mathsf{is\_clt}(T(j))$; or
(3) there exists T_m such that

$$T_m \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGT}), (\emptyset, \sigma_o, \circledcirc)]\!]\,,$$

and $\mathsf{get\_objevt}(T) = \mathsf{get\_objevt}(T_m)$.

For (1), by the operational semantics, we can prove $\mathsf{prog\text{-}t}(T)$ or $\mathsf{abt}(T)$ holds.
For (2), for any k and e, if $e \in \mathsf{pend\_inv}(T(1..k))$, since there exists $i > k$ such that $\mathsf{is\_clt}(T(i))$, by the operational semantics we know there exists j such that $k < j < i$ and $\mathsf{match}(e, T(j))$. Thus $\mathsf{prog\text{-}t}(T)$ holds.
For (3), by Lemma 11(1), there exists T_p such that

$$T_p \in \mathcal{T}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGTp1}), (\emptyset, \sigma_o, \circledcirc)]\!] \text{ and } T_p \backslash (\_, \textbf{out}, 1) = T.$$

Since $\Pi \sqsubseteq_\varphi^{1\omega} \Pi_A$, we know

$$\mathcal{O}_\omega[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGTp1}), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_\omega[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGTp1}), (\emptyset, \sigma_a, \circledcirc)]\!]\,.$$

From Lemma 12, we know $\mathsf{get\_obsv}(T_p)$ is an infinite trace of $(\_, \textbf{out}, 1)$. Thus $|T_p| = \omega$ and the following (B.15) holds:

$$\forall i.\ \exists j.\ j \geq i \wedge T_p(j) = (\_, \textbf{out}, 1)\,.$$

As in the proof of (B.12), we prove the following (B.16) from (B.15):

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T_p(j))\,.$$

Since $T_p \backslash (\_, \textbf{out}, 1) = T$, we know

$$\forall i.\ \exists j.\ j \geq i \wedge \mathsf{is\_ret}(T(j))\,.$$

Thus for any i and e, if $e \in \mathsf{pend\_inv}(T(1..i))$, then there exists $j > i$ such that $\mathsf{is\_ret}(T(j))$ holds. By the operational semantics and the generation of T, we know $\mathsf{match}(e, T(j))$ holds. Thus $\mathsf{prog\text{-}t}(T)$ holds. Then we are done.

Proofs of Theorem ??(2) We need to prove that if $\Pi \sqsubseteq_\varphi \Pi_A$ and $\mathsf{seq\text{-}term}_\varphi(\Pi)$, then for any C_1, σ_c, σ_o and σ_a such that $\varphi(\sigma_o) = \sigma_a$, we have

$$\mathcal{O}_\omega[\![(\textbf{let } \Pi \textbf{ in } C_1), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{O}_\omega[\![(\textbf{let } \Pi_A \textbf{ in } C_1), (\sigma_c, \sigma_a, \circledcirc)]\!]\,.$$

Thus we only need to prove: for any T,

(1) If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* \textbf{abort}$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* \textbf{abort}$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.
(2) If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^* (\textbf{skip}, \_)$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^* (\textbf{skip}, \_)$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.
(3) If $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$,
then there exists T_a such that
$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$ and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$.

(1) and (2) are proved in Lemma 13.

For (3), as in the proofs for (B.13), we define the simulation relation $\precsim$ in Figure 11(d), and prove the following (B.19):

> For any W, $\mathcal{S}$, W_1, $\mathcal{S}_1$, W_2, $\mathcal{S}_2$, W_3, $\mathcal{S}_3$, T_0 and T_1,
> if $(W, \mathcal{S})$ is well-formed and out of method calls, $(W, \mathcal{S}) \overset{T_0}{\longmapsto}{}^* (W_1, \mathcal{S}_1)$,
> $(W_1, \mathcal{S}_1) \precsim (W_2, \mathcal{S}_2; W_3, \mathcal{S}_3)$, $(W_1, \mathcal{S}_1) \overset{T_1}{\longmapsto}{}^\omega \cdot$ and $\mathsf{lock\text{-}free}(T_0 :: T_1)$,
> then there exists T_3 such that $(W_3, \mathcal{S}_3) \overset{T_3}{\longmapsto}{}^\omega \cdot$ and
> $T_1 \backslash (\_, \textbf{obj}) = T_3 \backslash (\_, \textbf{obj})$.

On the other hand, for any n, C_1, ..., C_n, σ_c, σ_o and σ_a, by Lemma 1, we know

$$\mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } C_1), (\sigma_c, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_1), (\emptyset, \sigma_o, \circledcirc)]\!] .$$

From $\Pi \sqsubseteq_\varphi \Pi_A$, by Lemma 3, we know $\Pi \subseteq_\varphi \Pi_A$. Thus, if $\varphi(\sigma_o) = \sigma_a$, then

$$\mathcal{H}[\![(\textbf{let } \Pi \textbf{ in } \mathsf{MGC}_1), (\emptyset, \sigma_o, \circledcirc)]\!] \subseteq \mathcal{H}[\![(\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_1), (\emptyset, \sigma_a, \circledcirc)]\!] .$$

Then we know

$$\begin{array}{l} (\textbf{let } \Pi \textbf{ in } C_1, (\sigma_c, \sigma_o, \circledcirc)) \\ \precsim (\textbf{let } \Pi_A \textbf{ in } \mathsf{MGC}_1, (\emptyset, \sigma_a, \circledcirc); \\ \quad \textbf{let } \Pi_A \textbf{ in } C_1, (\sigma_c, \sigma_a, \circledcirc)), \end{array}$$

Thus, if $(\lfloor \textbf{let } \Pi \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_o, \circledcirc)) \overset{T}{\longmapsto}{}^\omega \cdot$, by $\mathsf{seq\text{-}term}_\varphi(\Pi)$, we know $\mathsf{lock\text{-}free}(T)$. Then from (B.19) we get: there exists T_a such that

$$(\lfloor \textbf{let } \Pi_A \textbf{ in } C_1 \rfloor, (\sigma_c, \sigma_a, \circledcirc)) \overset{T_a}{\longmapsto}{}^\omega \cdot$$

and $\mathsf{get\_obsv}(T) = \mathsf{get\_obsv}(T_a)$, thus we are done.

A Separation Logic for Enforcing Declarative Information Flow Control Policies

David Costanzo and Zhong Shao

Yale University

Abstract. In this paper, we present a program logic for proving that a program does not release information about sensitive data in an unintended way. The most important feature of the logic is that it provides a formal security guarantee while supporting "declassification policies" that describe precise conditions under which a piece of sensitive data can be released. We leverage the power of Hoare Logic to express the policies and security guarantee in terms of state predicates. This allows our system to be far more specific regarding declassification conditions than most other information flow systems.
The logic is designed for reasoning about a C-like, imperative language with pointer manipulation and aliasing. We therefore make use of ideas from Separation Logic to reason about data in the heap.

1 Introduction

Information Flow Control (IFC) is a field of computer security concerned with tracking the propagation of information through a system. A primary goal of IFC reasoning is to formally prove that a system does not inadvertently leak high-security data to a low-security observer. A major challenge is to precisely define what "inadvertently" should mean here.

A simple solution to this challenge, taken by many IFC systems (e.g., [4, 5, 11, 16, 19]), is to define an information-release policy using a lattice of security labels. A *noninterference* property is imposed: information cannot flow down the lattice. Put another way, any data that the observer sees can only have been influenced by data with label less than or equal to the observer's label in the lattice. This property is sometimes called *pure noninterference*.

Purely-noninterfering systems are unfortunately not very useful. Almost all real-world systems need to violate noninterference sometimes. For example, consider one of the most standard security-sensitive situations: password authentication. In order for a password to be useful, there must be a way for a user to submit a guess at the password. If the guess is incorrect, then the user will be informed as such. However, the information that the guess was incorrect is dependent on the password itself; the user (who might be a malicious attacker) learns that the password is definitely *not* the one that was guessed. This represents a flow of information (albeit a minor one) from the high-security password to the low-security user, thus violating noninterference. In a purely noninterfering system, sensitive data has no way whatsoever of affecting the outcome of a

computation, and so the situation is essentially equivalent to the data not being present in the system at all.

There have been numerous attempts at refining the notion of inadvertent information release beyond the rules of a strict lattice structure. IFC systems commonly allow for some method of *declassification*, a term used to describe an information leak (i.e., an information flow moving down the security lattice) that is understood to be in some way "acceptable" or "purposeful" (as opposed to "inadvertent"). These declassifications violate the pure noninterference property described above. Ideally, an IFC system should still provide some sort of security guarantee even in the presence of declassification. It is quite rare, however, for a system to have a satisfactory formal guarantee. Those that do usually must make significant concessions that limit the generality and usefulness of the system.

Our goal is to leverage the strengths of a program logic to devise a powerful IFC system that provides formal security guarantees even in the presence of declassification. It turns out that we can use state predicates to refine the pure noninterference property into one that cleanly describes exactly how a piece of high-security data could affect observable output. Instead of simply saying that an observer cannot distinguish between any values of the high-security data, we say that the observer cannot distinguish between any values among a particular set — the set described by the state predicate.

Our contributions in this paper are as follows:

- We define a novel, security-aware semantics for a simple imperative language with pointer arithmetic and aliasing that tracks information flow through label propagation. We show that this semantics is sensible by relating its executions back to a standard (security-ignorant) small-step operational semantics.
- We present a program logic for formally verifying the safety of a program under the security-aware semantics. The logic builds on ideas from Hoare Logic [6] and Separation Logic [13, 14].
- We prove a strong security guarantee for any program that is verified using our program logic. This guarantee is a generalization of traditional pure noninterference.
- All of the technical work in this paper is fully formalized and proved in the Coq proof assistant.

The remainder of this paper is organized as follows: Section 2 informally discusses how our system works and highlights contributions; Section 3 defines our language, state model, and operational semantics; Section 4 describes the program logic and its soundness theorem relative to the operational semantics; Section 5 describes the noninterference-based security guarantee provided by the program logic; and Section 6 describes related work and concludes.

2 Informal Discussion

In this section, we will describe our system informally in order to provide some high-level motivation. We pick a starting point of a C-like, imperative language

with pointer arithmetic and aliasing, as we would like our logic to be applicable to low-level systems code. The main operations of our language are variable assignment $x := E$, heap dereference/load $x := [E]$, and heap dereference/store $[E] := E'$. The expressions E can be any standard mathematical expressions on program variables, so pointer arithmetic is allowed. Aliasing is also clearly allowed since $[x]$ and $[y]$ refer to the same heap location if x and y contain the same value.

2.1 Security Labels

Our language semantics will track information flow by attaching a *security label* to every value in the program state. For simplicity of presentation, we will assume that the only labels are `Lo` and `Hi` (a more general version of our system allows labels to be any set of elements that form a lattice structure). Unlike many IFC systems, we attach the label to the *value* rather than the *location*. This means that a program is allowed to, for example, overwrite some `Lo` data stored in variable x with some other `Hi` data. Many other systems would instead label the location x as `Lo`, meaning that `Hi` data could never be written into it. Supporting label overwrites allows our system to verify a wider variety of programs.

Label propagation is done in a mostly obvious way. If we have a direct assignment such as $x := y$, then the label of y's data propagates into x along with the data itself. We compute the composite label of an expression such as $2 * x + z$ to be the least upper bound of the labels of its constituent parts (for the two-element lattice of `Lo` and `Hi`, this will be `Lo` if and only if each constituent label is `Lo`). For the heap-read command $x := [E]$, we must propagate both the label of E and the label of the data located at heap address E into x. In other words, if we read some low-security data from the heap using a high-security pointer, the result must be tainted as high security in order for our information flow tracking to be accurate. Similarly, the heap-write command $[E] := E'$ must propagate both the label of E' and the label of pointer E into the location E in the heap. As a general rule for any of these atomic commands, we compute the composite label of the entire read-set, and propagate that into all locations in the write-set.

2.2 Noninterference

As discussed in Section 1, the ultimate goal of our IFC system is to prove a formal security guarantee that holds for any verified program. The standard security guarantee is noninterference, which says that the initial values of `Hi` data have no effect on the "observable behavior" of a program's execution. We choose to define observable behavior in terms of a special output channel. We include an output command in our language, and an execution's observable behavior is defined to be exactly the sequence of values that the execution outputs.

The standard way to express this noninterference property formally is in terms of two executions: a program is deemed to be noninterfering if two executions of the program from *observably equivalent* initial states always yield

identical outputs. Two states are defined to be observably equivalent when only their high-security values differ. Thus this property describes what one would expect: changing the value of any high-security data in the initial state will cause no change in the program's output.

One of our key insights is that this noninterference property can be refined by requiring a precondition to hold on the initial state of an execution. That is, we alter the property to say that two executions will yield identical outputs if they start from two observably equivalent states that both satisfy some state predicate P. This weakening of noninterference is interesting for two reasons. First, it provides a link between information flow security and Hoare Logic (a program logic that derives pre/postconditions as state predicates). Second, this property describes a certain level of dependency between high-security inputs and low-security outputs, rather than the complete independence of pure noninterference. This means that a program that satisfies this weaker noninterference may be semantically declassifying data. In this sense, we can use this property as an interesting security guarantee for a program that may declassify some data.

To better understand this weaker version of noninterference, let us consider a few examples.

Public Parity Suppose we have a variable x that contains some high-security data. We wish to specify a *declassification policy* which says that only the parity of the `Hi` value can be released to the public. We will accomplish this by verifying the security of some program with a precondition P that says "x contains high data, y contains low data, and $y = x\%2$". Our security property then says that if we have an execution from some state satisfying P, then changing the value of x will not affect the output as long as the new state also satisfies P. Since y is the parity of x and is unchanged in the two executions, this means that as long as we change x to some other value *that has the same parity*, the output will be unchanged. Indeed, this is exactly the property that one would expect to have with a policy that releases only the parity of a secret value: only the secret's parity can influence the observable behavior.

Public Average Suppose we have three secrets stored in x, y, and z, and we are only willing to release their average as public (e.g., the secrets are employee salaries at a particular company). This is similar to the previous example, except that we now have multiple secrets. The precondition P will say that x, y, and z all contain `Hi` data, a contains `Lo` data, and $a = (x + y + z)/3$. In this situation, noninterference will say that we can change the value of the *set* of secrets from any triple to any other triple, and the output will be unaffected as long as the average of the three values is unchanged.

Public Zero Suppose we have a a secret stored in x, and we are only willing to release it if it is zero. We could take the approach of the previous two examples and store a public boolean in another variable which is true if and only if x is 0. However, there is an even simpler way to represent the desired policy without using an extra variable. Our precondition P will say that either x is 0 and

its label is `Lo`, or x is nonzero and its label is `Hi`. This is an example of a *conditional label*: a label whose value depends on some state predicate. If x is 0, then noninterference says nothing since there is no high-security data in the state. If x is nonzero, then noninterference says that changing its value (but *not* its label) will have no effect on the output as long as P still holds; in order for P to still hold, we must be changing x to some other *nonzero* value. Hence all nonzero values of x will look the same to an observer. Conditional labels are a novelty of our system; we will see in Section 4 how they can be a powerful tool for verifying the security of a program.

3 Language and Semantics

Our programming language is defined as follows:

$$\begin{array}{rll} \text{(Exp)} & E & ::= x \mid c \mid E + E \mid \cdots \\ \text{(BExp)} & B & ::= \texttt{false} \mid E = E \mid B \wedge B \mid \cdots \\ \text{(Cmd)} & C & ::= \texttt{skip} \mid \texttt{output}\, E \mid x := E \mid x := [E] \mid [E] := E \mid C; C \\ & & \quad \mid \texttt{if}\, B\, \texttt{then}\, C\, \texttt{else}\, C \mid \texttt{while}\, B\, \texttt{do}\, C \end{array}$$

Valid code includes variable assignment, heap load/store, if statements, while loops, and output. Our model of a program state, consisting of a variable store and a heap, is given by:

$$\begin{array}{rll} \text{(Lbl)} & L & ::= \texttt{Lo} \mid \texttt{Hi} \\ \text{(Val)} & V & ::= \mathbb{Z} \times \text{Lbl} \\ \text{(Store)} & s & ::= \text{Var} \to \text{option Val} \\ \text{(Heap)} & h & ::= \mathbb{N} \to \text{option Val} \\ \text{(State)} & \sigma & ::= (s, h) \end{array}$$

Given a variable store s, we define a denotational semantics $[\![E]\!]s$ that evaluates an expression to a pair of integer and label, with the label being the least upper bound of the labels of the constituent parts. The denotation of an expression also may evaluate to `None`, indicating that the program state does not contain the necessary resources to evaluate. We have a similar denotational semantics for boolean expressions. The formal definitions of these semantics are omitted here as they are standard and straightforward. Note that we will sometimes write $[\![E]\!]\sigma$ as shorthand for $[\![E]\!]$ applied to the store of state σ.

Figure 1 defines our operational semantics. The semantics is security-aware, meaning that it keeps track of security labels on data and propagates these labels throughout execution in order to track which values might have been influenced by some high-security data. The semantics operates on machine configurations, which consist of program state, code, and a list of commands called the continuation stack (we use a continuation-stack approach solely for the purpose of simplifying some proofs). The transition arrow of the semantics is annotated with a *program counter label*, which is a standard IFC construct used to keep track of information flow resulting from the control flow of the execution. Whenever an

$$\frac{[\![E]\!]s = \texttt{Some}\ (n, l)}{\langle (s,h),\ x := E,\ K\rangle \xrightarrow[l']{} \langle (s[x \mapsto (n, l \sqcup l')], h),\ \texttt{skip},\ K\rangle}\ (\text{ASSGN})$$

$$\frac{[\![E]\!]s = \texttt{Some}\ (n_1, l_1) \qquad h(n_1) = \texttt{Some}\ (n_2, l_2)}{\langle (s,h),\ x := [E],\ K\rangle \xrightarrow[l']{} \langle (s[x \mapsto (n_2, l_1 \sqcup l_2 \sqcup l')], h),\ \texttt{skip},\ K\rangle}\ (\text{READ})$$

$$\frac{[\![E]\!]s = \texttt{Some}\ (n_1, l_1) \qquad h(n_1) \neq \texttt{None} \qquad [\![E']\!]s = \texttt{Some}\ (n_2, l_2)}{\langle (s,h),\ [E] := E',\ K\rangle \xrightarrow[l']{} \langle (s, h[n_1 \mapsto (n_2, l_1 \sqcup l_2 \sqcup l')]),\ \texttt{skip},\ K\rangle}\ (\text{WRITE})$$

$$\frac{[\![E]\!]\sigma = \texttt{Some}\ (n, \texttt{Lo})}{\langle \sigma,\ \texttt{output}\ E,\ K\rangle \xrightarrow[\texttt{Lo}]{[n]} \langle \sigma,\ \texttt{skip},\ K\rangle}\ (\text{OUTPUT})$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{true}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ K\rangle \xrightarrow[l']{} \langle \sigma,\ C_1,\ K\rangle}\ (\text{IF-TRUE})$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{false}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ K\rangle \xrightarrow[l']{} \langle \sigma,\ C_2,\ K\rangle}\ (\text{IF-FALSE})$$

$$\frac{\begin{array}{c}[\![B]\!]\sigma = \texttt{Some}\ (\_, \texttt{Hi}) \\ \langle \texttt{mark\_vars}(\sigma, \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2),\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ []\rangle \xrightarrow[\texttt{Hi}]{}_n \langle \sigma',\ \texttt{skip},\ []\rangle\end{array}}{\langle \sigma,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ K\rangle \xrightarrow[\texttt{Lo}]{} \langle \sigma',\ \texttt{skip},\ K\rangle}\ (\text{IF-HI})$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{true}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle \xrightarrow[l']{} \langle \sigma,\ C; \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle}\ (\text{WHILE-TRUE})$$

$$\frac{[\![B]\!]\sigma = \texttt{Some}\ (\texttt{false}, l) \qquad l \sqsubseteq l'}{\langle \sigma,\ \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle \xrightarrow[l']{} \langle \sigma,\ \texttt{skip},\ K\rangle}\ (\text{WHILE-FALSE})$$

$$\frac{\begin{array}{c}[\![B]\!]\sigma = \texttt{Some}\ (\_, \texttt{Hi}) \\ \langle \texttt{mark\_vars}(\sigma, \texttt{while}\ B\ \texttt{do}\ C),\ \texttt{while}\ B\ \texttt{do}\ C,\ []\rangle \xrightarrow[\texttt{Hi}]{}_n \langle \sigma',\ \texttt{skip},\ []\rangle\end{array}}{\langle \sigma,\ \texttt{while}\ B\ \texttt{do}\ C,\ K\rangle \xrightarrow[\texttt{Lo}]{} \langle \sigma',\ \texttt{skip},\ K\rangle}\ (\text{WHILE-HI})$$

$$\frac{}{\langle \sigma,\ C_1; C_2,\ K\rangle \xrightarrow[l]{} \langle \sigma,\ C_1,\ C_2 :: K\rangle}\ (\text{SEQ})$$

$$\frac{}{\langle \sigma,\ \texttt{skip},\ C :: K\rangle \xrightarrow[l]{} \langle \sigma,\ C,\ K\rangle}\ (\text{SKIP}) \qquad \frac{}{\langle \sigma,\ C,\ K\rangle \xrightarrow[l]{}_0 \langle \sigma,\ C,\ K\rangle}\ (\text{ZERO})$$

$$\frac{\langle \sigma,\ C,\ K\rangle \xrightarrow[l]{o} \langle \sigma',\ C',\ K'\rangle \qquad \langle \sigma',\ C',\ K'\rangle \xrightarrow[l]{o'}_n \langle \sigma'',\ C'',\ K''\rangle \qquad n > 0}{\langle \sigma,\ C,\ K\rangle \xrightarrow[l]{o +\!\!+ o'}_{n+1} \langle \sigma'',\ C'',\ K''\rangle}\ (\text{SUCC})$$

Fig. 1. Security-Aware Operational Semantics

execution enters a conditional construct, it raises the pc label by the label of the boolean expression evaluated; the pc label then taints any assignments that are made within the conditional construct. The transition arrow is also annotated with a list of outputs (equal to the empty list when not explicitly written) and the number of steps (equal to 1 when not explicitly written).

Two of the rules for conditional constructs make use of a function called `mark_vars`. The function $\texttt{mark\_vars}(\sigma, C)$ alters σ by setting the label of each variable in $\texttt{modifies}(C)$ to `Hi`, where $\texttt{modifies}(C)$ is a standard syntactic function returning an overapproximation of the store variables that may be modified by C. Thus, whenever we raise the pc label to `Hi`, our semantics taints all store variables that appear on the left-hand side of an assignment or heap-read command within the conditional construct, even if some of these commands do not actually get executed. Note that regardless of which branch of an if statement is taken, the semantics taints all the variables in *both* branches. This is required for noninterference, due to the well-known fact that the *lack* of assignment in a branch of an if statement can leak information about the branching expression. Consider, for example, the following program:

```
1    y := 1;
2    if (x = 0) then y := 0 else skip;
3    if (y = 0) then skip else output 1;
```

Suppose x contains `Hi` data initially, while y contains `Lo` data. If x is 0, then y will be assigned 0 at line 2 and tainted with a `Hi` label (by the pc label). Then nothing happens at line 3, and the program produces no output. If x is nonzero, however, nothing happens at line 2, so y still has a `Lo` label at line 3. Thus the output command at line 3 executes without issue. Therefore the output of this program depends on the `Hi` data in x, even though our instrumented semantics executes safely. We choose to resolve this issue by using the `mark_vars` function in the semantics. Then y will be tainted at line 2 regardless of the value of x, and so the semantics will get stuck at line 3 when x is nonzero. In other words, we would only be able to verify this program with a precondition saying that $x = 0$ — the program is indeed noninterfering with respect to this precondition (according to our generalized noninterference definition described in Section 2).

The operational semantics presented here is mixed-step and manipulates security labels directly. In order make sense of such a non-standard semantics, we need to relate it in some way to a standard one. A standard, single-step semantics is defined in the Appendix. This semantics operates on states without labels, and it does not use continuation stacks. Given a state σ with labels, we write $\bar{\sigma}$ to represent the same state with all labels erased from both the store and heap. We will also use τ to range over states without labels. Then the following two theorems hold:

Theorem 1. *Suppose* $\langle \sigma, C, [] \rangle \xrightarrow{o}_* \langle \sigma', \texttt{skip}, [] \rangle$ *in the instrumented semantics. Then, for some* τ, $\langle \bar{\sigma}, C \rangle \xrightarrow{o}_* \langle \tau, \texttt{skip} \rangle$ *in the standard semantics.*

$$
\begin{array}{rcl}
P, Q & ::= & \texttt{emp} \mid E \mapsto \_ \mid E \mapsto (n, l) \mid B \mid x.\texttt{lbl} = l \mid x.\texttt{lbl} \sqsubseteq l \\
& & \mid \texttt{lbl}(E) = l \mid \exists X \,.\, P \mid P \wedge Q \mid P \vee Q \mid P * Q
\end{array}
$$

$$
\begin{array}{rcl}
[\![P]\!] & : & \mathcal{P}(\texttt{state}) \\
(s,h) \in [\![\texttt{emp}]\!] & \iff & h = \emptyset \\
(s,h) \in [\![E \mapsto \_]\!] & \iff & \exists a, n, l \,.\, [\![E]\!]s = \texttt{Some}\ a \wedge h = [a \mapsto (n,l)] \\
(s,h) \in [\![E \mapsto (n,l)]\!] & \iff & \exists a \,.\, [\![E]\!]s = \texttt{Some}\ a \wedge h = [a \mapsto (n,l)] \\
(s,h) \in [\![B]\!] & \iff & [\![B]\!]s = \texttt{Some true} \\
(s,h) \in [\![x.\texttt{lbl} = l]\!] & \iff & \exists n \,.\, s(x) = \texttt{Some}\ (n,l) \\
(s,h) \in [\![x.\texttt{lbl} \sqsubseteq l]\!] & \iff & \exists n, l' \,.\, s(x) = \texttt{Some}\ (n,l') \text{ and } l' \sqsubseteq l \\
(s,h) \in [\![\texttt{lbl}(E) = l]\!] & \iff & \bigsqcup_{x \in \texttt{vars}(E)} \mathrm{snd}(s(x)) = l \\
(s,h) \in [\![\exists X \,.\, P]\!] & \iff & \exists v \in \mathbb{Z} + \mathrm{Lbl} \,.\, (s,h) \in [\![P[v/X]]\!] \\
(s,h) \in [\![P \wedge Q]\!] & \iff & (s,h) \in [\![P]\!] \cap [\![Q]\!] \\
(s,h) \in [\![P \vee Q]\!] & \iff & (s,h) \in [\![P]\!] \cup [\![Q]\!] \\
(s,h) \in [\![P * Q]\!] & \iff & \left(\begin{array}{l} \exists h_0, h_1 \,.\, h_0 \uplus h_1 = h \\ \text{and } (s,h_0) \in [\![P]\!] \\ \text{and } (s,h_1) \in [\![Q]\!] \end{array} \right)
\end{array}
$$

Fig. 2. Assertion Syntax and Semantics

Theorem 2. *Suppose* $\langle \bar{\sigma}, C \rangle \xrightarrow{o}_* \langle \tau, \texttt{skip} \rangle$ *in the standard semantics, and suppose* $\langle \sigma, C, [] \rangle$ *never gets stuck when executed in the instrumented semantics. Then, for some* σ', $\langle \sigma, C, [] \rangle \xrightarrow{o}_* \langle \sigma', \texttt{skip}, [] \rangle$ *in the instrumented semantics.*

These theorems together guarantee that the two semantics produce identical observable behaviors (outputs) on terminating executions, as long as the instrumented semantics does not get stuck. Our program logic will of course guarantee that the instrumented semantics does not get stuck in any execution satisfying the precondition.

4 The Program Logic

In this section, we will present the logic that we use for verifying the security of a program. A logic judgment takes the form $l \vdash \{P\}\, C\, \{Q\}$. P and Q are the pre- and postconditions, C is the program to be executed, and l is the pc label under which the program is verified. P and Q are *state assertions*, whose syntax and semantics are given in Figure 2.

Note We allow assertions to contain logical variables, but we elide the details here to avoid complicating the presentation. In Figure 2, we claim that the type

of $[\![P]\!]$ is a set of states — in reality, the type is a function from logical variable environments to sets of states. In an assertion like $E \mapsto (n, l)$, the n and l may be logical variables rather than constants.

Definition 1 (Sound judgment). *We say that a judgment* $l \vdash \{P\}\, C\, \{Q\}$ *is* sound *if, for any state* $\sigma \in [\![P]\!]$*, the following two properties hold:*

1. *The operational semantics cannot get stuck when executed from initial configuration* $\langle \sigma, C, [] \rangle$ *under context* l*.*
2. *If the operational semantics executes from initial configuration* $\langle \sigma, C, [] \rangle$ *under context* l *and terminates at state* σ'*, then* $\sigma' \in [\![Q]\!]$*.*

Selected inference rules for our logic are shown in Figure 3. The rules make use of two auxiliary syntactic functions, $P \backslash x$ and $P \backslash x.\texttt{lbl}$ (read the backslash operator as "delete"). $P \backslash x$ replaces any atomic assertions within P referring to x by the assertion `true`. Similarly, $P \backslash x.\texttt{lbl}$ replaces atomic assertions referring to $x.\texttt{lbl}$ by `true`. We also sometimes abuse notation and write $P \backslash S$ or $P \backslash S.\texttt{lbl}$, where S is a set of variables, to indicate the iterative folding of these functions over the set S. The important fact about these auxiliary functions is that, if P holds on some state and we perform an assigment into x, then $P \backslash x$ will hold on the resulting state. Furthermore, if we change only the label of x without touching its data (this is done by the `mark_vars` function described in Section 3), then $P \backslash x.\texttt{lbl}$ will hold on the resulting state.

Here are a few interesting points to note about these inference rules:

- While the rules shown here mostly involve detailed reasoning about label propagation, we can also prove the soundness of simpler versions of the rules that do not reason about labels and, consequentially, do not have any label-related proof obligations.
- The (IF) and (WHILE) rules may look rather complex, but almost all of that is just describing how to reason about the `mark_vars` function that gets applied at the beginning of a conditional construct when the pc label increases.
- An additional complexity present in the (IF) rule involves the labels l_t and l_f. In fact, these labels describe a novel and interesting feature of our system: when verifying an if statement, it might be possible to reason that the pc label gets raised by l_t in one branch and by l_f in the other, based on the fact that B holds in one branch but not in the other. This is interesting if l_t and l_f are different labels. In every other static-analysis IFC system we are aware of, a particular pc label must be determined at the entrance to the conditional, and this pc label will propagate to both branches. We will provide an example program later in this section that illustrates this novelty.

Given our logic inference rules, we can prove the following theorem:

Theorem 3 (Soundness). *If* $l \vdash \{P\}\, C\, \{Q\}$ *is derivable according to our inference rules, then it is a* sound judgment*, as defined in Definition 1.*

$$\texttt{mark\_vars}(P, S, l, l') \triangleq \begin{cases} P & , \quad \text{if } l \sqsubseteq l' \\ P \backslash S.\texttt{lbl} \wedge \left(\bigwedge_{x \in S} l \sqcup l' \sqsubseteq x.\texttt{lbl} \right) & , \quad \text{otherwise} \end{cases}$$

$$\frac{}{l \vdash \{P\}\, \texttt{skip}\, \{P\}}\ (\text{SKIP}) \qquad \frac{P \Rightarrow \texttt{lbl}(E) = \texttt{Lo}}{\texttt{Lo} \vdash \{P\}\, \texttt{output}\, E\, \{P\}}\ (\text{OUTPUT})$$

$$\frac{P \Rightarrow \texttt{lbl}(E) = l}{l' \vdash \{P\}\, x := E\, \{(P \backslash x)[E/x] \wedge x.\texttt{lbl} = l \sqcup l'\}}\ (\text{ASSIGN})$$

$$\frac{P \Rightarrow \texttt{lbl}(E) = l_1 \qquad P \Rightarrow E \mapsto (n, l_2)}{l \vdash \{P\}\, x := [E]\, \{P \backslash x \wedge x = n \wedge x.\texttt{lbl} = l_1 \sqcup l_2 \sqcup l\}}\ (\text{READ})$$

$$\frac{P \Rightarrow \texttt{lbl}(E) = l_1 \qquad P \Rightarrow \texttt{lbl}(E') = l_2 \qquad P \Rightarrow E \mapsto \_}{l \vdash \{P\}\, [E] := E'\, \{P \wedge \exists n\, .\, E \mapsto (n, l_1 \sqcup l_2 \sqcup l) \wedge E' = n\}}\ (\text{WRITE})$$

$$\frac{\begin{array}{c} B \wedge P \Rightarrow \texttt{lbl}(B) = l_t \\ \neg B \wedge P \Rightarrow \texttt{lbl}(B) = l_f \qquad S = \texttt{modifies}(\texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2) \\ l_t \sqcup l' \vdash \{B \wedge \texttt{mark\_vars}(P, S, l_t, l')\}\, C_1\, \{Q\} \\ l_f \sqcup l' \vdash \{\neg B \wedge \texttt{mark\_vars}(P, S, l_f, l')\}\, C_2\, \{Q\} \end{array}}{l' \vdash \{P\}\, \texttt{if}\, B\, \texttt{then}\, C_1\, \texttt{else}\, C_2\, \{Q\}}\ (\text{IF})$$

$$\frac{\begin{array}{c} P \Rightarrow \texttt{lbl}(B) = l \qquad S = \texttt{modifies}(\texttt{while}\, B\, \texttt{do}\, C) \\ l \sqcup l' \vdash \{B \wedge \texttt{mark\_vars}(P, S, l, l')\}\, C\, \{\texttt{mark\_vars}(P, S, l, l')\} \end{array}}{l' \vdash \{P\}\, \texttt{while}\, B\, \texttt{do}\, C\, \{\neg B \wedge \texttt{mark\_vars}(P, S, l, l')\}}\ (\text{WHILE})$$

$$\frac{l \vdash \{P\}\, C_1\, \{Q\} \qquad l \vdash \{Q\}\, C_2\, \{R\}}{l \vdash \{P\}\, C_1; C_2\, \{R\}}\ (\text{SEQ})$$

$$\frac{P' \Rightarrow P \qquad Q \Rightarrow Q' \qquad l \vdash \{P\}\, C\, \{Q\}}{l \vdash \{P'\}\, C\, \{Q'\}}\ (\text{CONSEQ})$$

$$\frac{l \vdash \{P_1\}\, C\, \{Q_1\} \qquad l \vdash \{P_2\}\, C\, \{Q_2\}}{l \vdash \{P_1 \wedge P_2\}\, C\, \{Q_1 \wedge Q_2\}}\ (\text{CONJ})$$

$$\frac{l \vdash \{P\}\, C\, \{Q\} \qquad \texttt{modifies}(C) \cap \texttt{vars}(R) = \emptyset}{l \vdash \{P * R\}\, C\, \{Q * R\}}\ (\text{FRAME})$$

Fig. 3. Selected Inference Rules for the Logic

```
1  i := 0;
2  while (i < 64) do
3      x := [A+i];
4      if (x = 0)
5          then
6              output i
7          else
8              skip;
9      i := i+1
```

Fig. 4. Example: Alice's Private Calendar

We will not go over the proof of this theorem here since there is not really anything novel about it in regards to security. The proof is relatively straightforward and not significantly different from soundness proofs in other Hoare/separation logics. The primary theorem in this work is the one that says that any verified program satisfies our noninterference property — this will be discussed in detail in Section 5.

4.1 Example: Alice's Calendar

In the remainder of this section, we will show how our logic can be used to verify an interesting example. Figure 4 shows a program that we would like to prove is secure. Alice owns a calendar with 64 time slots beginning at some location designated by constant A. Each time slot is either 0 if she is free at that time, or some nonzero value representing an event if she is busy. Alice decides that all free time slots in her calendar should be considered low security, while the time slots with events should be secret. This policy allows for others to schedule a meeting time with her, as they can determine when she is available. Indeed, the example program shown here prints out all free time slots.

Figure 5 gives an overview of the verification, omitting a few trivial details. In between each line of code, we show the current pc label and a state predicate that currently holds. The program is verified with respect to Alice's policy, described by the precondition P defined in the figure. This precondition is the iterated separating conjunction of 64 calendar slots; each slot's label is `Lo` if its value is 0 and `Hi` otherwise. A major novelty of this verification regards the conditional statement at lines 4-8. As mentioned earlier, in other IFC systems, the label of the boolean expression "$x = 0$" would have to be determined at the time of entering the conditional, and its label would then propagate into both branches via the pc label. In our system, however, we can reason that the expression's label (and hence the resulting pc label) will be different depending on which branch is taken. If the "true" branch is taken, then we know that x is 0, and hence we know from the state assertion that its label is `Lo`. This means that the pc label is `Lo`, and so the output statement within this branch will not leak high-security data. If the "false" branch is taken, however, then we can reason that the pc label will be `Hi`, meaning that an output statement could result in

$$P \stackrel{\triangle}{=} \circledast_{i=0}^{63} \ (A + i \mapsto (n_i, l_i) \wedge n_i = 0 \iff l_i = \texttt{Lo})$$

```
    Lo ⊢ {P}
1   i := 0;
    Lo ⊢ {P ∧ 0 ≤ i ∧ i.lbl = Lo}
2   while (i < 64) do
        Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
3       x := [A+i];
        Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                    (x = 0 ⟺ x.lbl = Lo)}
4       if (x = 0)
5           then
                Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x = 0 ∧ x.lbl = Lo}
6               output i
                Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x = 0 ∧ x.lbl = Lo}
                Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
7           else
                Hi ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x ≠ 0 ∧ x.lbl = Hi}
8               skip;
                Hi ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo ∧
                              x ≠ 0 ∧ x.lbl = Hi}
                Hi ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
        Lo ⊢ {P ∧ 0 ≤ i < 64 ∧ i.lbl = Lo}
9       i := i+1
        Lo ⊢ {P ∧ 0 ≤ i ∧ i.lbl = Lo}
    Lo ⊢ {P ∧ i ≥ 64 ∧ 0 ≤ i ∧ i.lbl = Lo}
```

Fig. 5. Calendar Example Verification

a leaky program (e.g., if the value of x were printed). This program does not attempt to output anything within this branch, so it is still valid.

Since the program is verified with respect to precondition P, the noninterference guarantee for this example says that if we change any high-security event in Alice's calendar to any other high-security event (i.e., nonzero value), then the output will be unaffected. In other words, an observer cannot distinguish between any two events occurring at a particular time slot. This seems like exactly the property Alice would want to have, given that her policy specifies that all free slots are `Lo` and all events are `Hi`.

Aside on Completeness Our system is not complete — there are plenty of programs that are noninterfering with respect to some precondition, but cannot be verified in our logic using that precondition. For example, if we slightly modify the program of Figure 4 by changing line 8 to output i, then the program will always output all the numbers from 0 to 63 in order, regardless of values of high-security data. We would not be able to verify the program, however, because the pc label is `Hi` at line 8 and thus disallows any output. Interestingly, we have found in our experience that we can always rewrite a secure-but-unverifiable program in such a way that it produces the same output and becomes verifiable. For this example, it suffices to rewrite the program to simply print out the numbers 0 through 63 (without branching on elements in Alice's calendar).

A rather more complex example can be obtained by swapping lines 6 and 8 in the code of Figure 4. This program prints out all the time slots that are *not* free. Changing any (nonzero) event to any other (nonzero) event will not change this output, so the program is still secure with respect to Alice's policy. It is not verifiable for the same reason as before — output is disallowed at line 8. Nevertheless, this program can be rewritten in the following way (assume we add to the precondition that we have a 64-element array filled with `Lo` 0's, starting at location B):

```
1  i := 0;
2  while (i < 64) do
3      x := [A+i];
4      if (x = 0) then [B+i] := 1 else skip;
5      i := i+1;
6  i := 0;
7  while (i < 64) do
8      x := [B+i];
9      if (x = 0) then output i else skip;
10     i := i+1;
```

The ability to rewrite these safe-but-unverifiable programs is a completely informal observation we have made. A formal result is beyond the scope of this paper, but we hope to explore it in future work.

5 Noninterference

In this section, we will discuss the method for formally proving our system's security guarantee. Much of the work has already been done through careful design of the security-aware semantics and the inference rules of the program logic. The fundamental idea is that we can find a bisimulation relation for our `Lo`-context instrumented semantics. This relation will guarantee that two executions operate in lock-step, always producing the same program continuation and output.

The bisimulation relation we will use is called *observable equivalence*. It intuitively says that the low-security portions of two states are identical; the relation is commonly used in many IFC systems as a tool for proving noninterference. In our system, states σ_1 and σ_2 are observably equivalent if: (1) they contain equal values at all locations that are present and `Lo` in both states; and (2) the presence and labels of all store variables are the same in both states. This may seem like a rather odd notion of equivalence (in fact, it is not even transitive, so "equivalence" is a misnomer here) — two states can be observably equivalent even if some heap location contains `Hi` data in one state and `Lo` data in the other. To see why we need to define observable equivalence in this way, consider a heap-write command $[x] := E$ where x is a `Hi` pointer. If we vary the value of x, then we will end up writing to two different locations in the heap. Suppose we write to location 100 in one execution and location 200 in the other. Then location 100 will contain `Hi` data in the first execution (as the `Hi` pointer taints the value written), but it may contain `Lo` data in the second since we never wrote to it. Thus we design observable equivalence so that this situation is allowed.

The following definitions describe observable equivalence formally:

Definition 2 (Observable Equivalence of Stores). *Suppose s_1 and s_2 are variable stores. We say that they are observably equivalent, written $s_1 \sim s_2$, if, for all program variables x:*

- *If $s_1(x) = \texttt{None}$, then $s_2(x) = \texttt{None}$.*
- *If $s_1(x) = \texttt{Some}\ (v_1, \texttt{Hi})$, then $s_2(x) = \texttt{Some}\ (v_2, \texttt{Hi})$ for some v_2.*
- *If $s_1(x) = \texttt{Some}\ (v, \texttt{Lo})$, then $s_2(x) = \texttt{Some}\ (v, \texttt{Lo})$.*

Definition 3 (Observable Equivalence of Heaps). *Suppose h_1 and h_2 are heaps. We say that they are observably equivalent, written $h_1 \sim h_2$, if, for all natural numbers n:*

- *If $h_1(n) = \texttt{Some}\ (v_1, \texttt{Lo})$ and $h_2(n) = \texttt{Some}\ (v_2, \texttt{Lo})$, then $v_1 = v_2$.*

We say that two states are observably equivalent (written $\sigma_1 \sim \sigma_2$) when both their stores and heaps are observably equivalent. Given this definition, we define a convenient relational denotational semantics for state assertions as follows:

$$(\sigma_1, \sigma_2) \in [\![P]\!]^2 \iff \sigma_1 \in [\![P]\!] \wedge \sigma_2 \in [\![P]\!] \wedge \sigma_1 \sim \sigma_2$$

In order to state noninterference cleanly, it helps to define a "bisimulation semantics" consisting of the following single rule (the side condition will be discussed below):

$$\frac{\langle \sigma_1, C, K\rangle \xrightarrow[\texttt{Lo}]{o} \langle \sigma_1', C', K'\rangle \quad \langle \sigma_2, C, K\rangle \xrightarrow[\texttt{Lo}]{o} \langle \sigma_2', C', K'\rangle \qquad \text{(side condition)}}{\langle \sigma_1, \sigma_2, C, K\rangle \longrightarrow \langle \sigma_1', \sigma_2', C', K'\rangle}$$

Note that this bisimulation semantics operates on configurations consisting of a *pair* of states and a program. With this definition, we can split noninterference into the following progress and preservation properties.

Theorem 4 (Progress). *Suppose we derive* $\texttt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. For any* $(\sigma_1, \sigma_2) \in [\![P]\!]^2$*, suppose we have*

$$\langle \sigma_1, \sigma_2, C, K\rangle \longrightarrow_* \langle \sigma_1', \sigma_2', C', K'\rangle,$$

where $\sigma_1' \sim \sigma_2'$ *and* $(C', K') \neq (\texttt{skip}, [])$*. Then there exist* σ_1'', σ_2'', C'', K'' *such that*

$$\langle \sigma_1', \sigma_2', C', K'\rangle \longrightarrow \langle \sigma_1'', \sigma_2'', C'', K''\rangle$$

Theorem 5 (Preservation). *Suppose we have* $\sigma_1 \sim \sigma_2$ *and* $\langle \sigma_1, \sigma_2, C, K\rangle \longrightarrow \langle \sigma_1', \sigma_2', C', K'\rangle$*. Then* $\sigma_1' \sim \sigma_2'$*.*

For the most part, the proofs of these theorems are relatively straightforward. Preservation requires proving the following two simple lemmas about `Hi`-context executions:

1. `Hi`-context executions never produce output.
2. If the initial and final values of some location differ across a `Hi`-context execution, then the location must have a `Hi` label in the final state.

There is one significant difficulty in the proof that requires discussion. If C is a heap-read command $x := [E]$, then Preservation does not obviously hold. The reason for this comes from our odd definition of observable equivalence; in particular, the requirements for a heap location to be observably equivalent are weaker than those for a store variable. Yet the heap-read command is copying directly from the heap to the store. In more concrete terms, the heap location pointed to by E might have a `Hi` label in one state and `Lo` label in the other; but this means x will now have different labels in the two states, violating the definition of observable equivalence for the store.

We resolve this difficulty via the side condition in the bisimulation semantics. The side condition says that the situation we just described does not happen. More formally, it says that if C has the form $x := [E]$, then the heap location pointed to by E in σ_1 has the same label as the heap location pointed to by E in σ_2.

This side condition is sufficient for proving Preservation. However, we still need to show that the side condition holds in order to prove Progress. This fact comes from induction over the specific inference rules of our logic. For example, consider the (READ) rule from Section 4:

$$\frac{P \Rightarrow \texttt{lbl}(E) = l_1 \qquad P \Rightarrow E \mapsto (n, l_2)}{l \vdash \{P\}\, x := [E]\, \{P \backslash x \wedge x = n \wedge x.\texttt{lbl} = l_1 \sqcup l_2 \sqcup l\}} \text{ (READ)}$$

In order to use this rule, we are required to show that the precondition implies $E \mapsto (n, l_2)$. Since both states σ_1 and σ_2 satisfy the precondition, we see that the heap locations pointed to by E both have label l_2, and so the side condition holds. Note that the side condition holds even if l_2 is a logical variable rather than a constant.

In order to prove that the side condition holds for *every* verified program, we need to show it holds for all inference rules involving a heap-read command. In particular, this means that no heap-read rule in our logic can have a precondition that only implies $E \mapsto \_$.

Now that we have the Progress and Preservation theorems, we can easily combine them to prove the overall noninterference theorem for our instrumented semantics:

Theorem 6 (Noninterference, Instrumented Semantics). *Suppose we derive* $\texttt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. Pick any state* $\sigma_1 \in [\![P]\!]$*, and consider changing the values of any* `Hi` *data in* σ_1 *to obtain some* $\sigma_2 \in [\![P]\!]$*. Suppose, in the instrumented semantics, we have*

$$\langle \sigma_1,\, C,\, [] \rangle \xrightarrow[\texttt{Lo}]{o_1}{}_* \langle \sigma_1',\, \texttt{skip},\, [] \rangle$$

and

$$\langle \sigma_2,\, C,\, [] \rangle \xrightarrow[\texttt{Lo}]{o_2}{}_* \langle \sigma_2',\, \texttt{skip},\, [] \rangle.$$

Then $o_1 = o_2$*.*

Finally, we can use the results from Section 3 along with the safety guaranteed by our logic to prove the final, end-to-end noninterference theorem:

Theorem 7 (Noninterference, True Semantics). *Suppose we derive* $\texttt{Lo} \vdash \{P\}\, C\, \{Q\}$ *using our program logic. Pick any state* $\sigma_1 \in [\![P]\!]$*, and consider changing the values of any* `Hi` *data in* σ_1 *to obtain some* $\sigma_2 \in [\![P]\!]$*. Suppose, in the true semantics, we have*

$$\langle \bar{\sigma}_1,\, C \rangle \xrightarrow{o_1}{}_* \langle \tau_1,\, \texttt{skip} \rangle$$

and

$$\langle \bar{\sigma}_2,\, C \rangle \xrightarrow{o_2}{}_* \langle \tau_2,\, \texttt{skip} \rangle.$$

Then $o_1 = o_2$*.*

6 Related Work

There are many different systems for reasoning about information flow. We will briefly discuss some of the more closely-related ones here.

Some IFC systems with declassification, such as Hi-Star [22], Flume [8], and RESIN [21], reason at the operating system or process level, rather than the language level. These systems can support complex security policies, but their formal guarantees suffer due to how coarse-grained they are.

On the language-level side of IFC [15], there are many type systems and program logics that share similarities with our logic.

Amtoft et al. [1] develop a program logic for proving noninterference of a program written in a simple object-oriented language. They use relational assertions of the form "x is independent from high-security data." Such an assertion is equivalent to saying that x contains `Lo` data in our system. Thus their logic can be used to prove that the final values of low-security data are independent from initial values of high-security data — this is pure noninterference. Note that, unlike our system, theirs does not attempt to reason about declassification. This is the primary advantage of our system over theirs. Some other differences between these IFC systems are:

- We allow pointer arithmetic, while they disallow it by using an object-oriented language. Pointer arithmetic adds significant complexity to information flow reasoning. In particular, their system uses a technique similar to our `mark_vars` function for reasoning about conditional constructs, except that they syntactically check for all locations in both the store and heap that might be modified within the conditional. With the arbitrary pointer arithmetic of our system, it is not possible to syntactically bound which heap locations will be written to, so we require the additional semantic technique described in Section 5 that involves enforcing a side condition on the bisimulation semantics.
- Our model of observable behavior provides some extra leniency in verification. Our system allows bad leaks to happen within the program state, so long as these leaks are not made observable via an output command. In their system (and most other IFC systems), the enforcment mechanism must prevent those leaks within program state from happening in the first place.

Banerjee et al. [3] develop an IFC system that specifies declassification policies through state predicates in basically the same way that we do. For example, they might have a (relational) precondition of "$A(x \geq y)$," saying that two states agree on the truth value of $x \geq y$. This corresponds directly to a precondition of "$x \geq y$" in our system, and security guarantees for the two systems are both stated relative to the precondition. The two systems have very similar goals, but there are a number of significant differences in the basic setup that make the systems quite distinct:

- Their system does not attempt to reason about the program heap at all. They have some high-level discussions about how one might support pointers in their setup, but there is nothing formal.

- Their system enforces noninterference primarily through a type system (rather than a program logic). The declassification policies, specified by something similar to a Hoare triple, are only used at specific points in the program where explicit "declassify" commands are executed. A type system enforces pure noninterference for the rest of the program besides the declassify commands. Their end-to-end security guarantee then talks about how the knowledge of an observer can only increase at those points where a declassify command is executed (a property known in the literature as "gradual release"). Thus their security guarantee for individual declassification commands looks very similar to our version of noninterference, but their end-to-end security guarantee looks quite different. We do not believe that there is any comparable notion of gradual release in our system, as we do not have explicit program points where declassification occurs.
- Because they use a type system, their system must statically pick security labels for each program variable. This means that there is no notion of dynamically propagating labels during execution, nor is there any way to express our novel concept of conditional labels. As a result, the calendar example program of Section 4 would not be verifiable in their system.

Delimited Release [16] is an IFC system that allows certain prespecified expressions (called *escape hatches*) to be declassified. For example, a declassification policy for high-security variable h might say that the expression $h\%2$ should be considered low security. Relaxed Noninterference [9] uses a similar idea, but builds a lattice of semantic declassification policies, rather than syntactic escape hatches — e.g., h would have a policy of $\lambda x \,.\, x\%2$. Our system can easily express any policy from these systems, using a precondition saying that some low-security data is equal to the escape hatch function applied to the secret data. Our strong security guarantee is identical to the formal guarantees of both of these systems, saying that the high-security value will not affect the observable behavior as long as the escape hatch valuation is unchanged.

Relational Hoare Type Theory (RHTT) [12] is a logic framework for verifying information-flow properties. It is based on a highly general relational logic. The system can be used to reason about a wide variety of security-related notions, including declassification, information erasure, and state-dependent access control. One advantage of our system over RHTT is that we have fine-tuned our system for reasoning about noninterference. A program verification in our system requires a relatively small amount of work, since much of the noninterference proof is already handled by the framework. RHTT, on the other hand, is extremely general to the point that if you want to prove an information flow property on a program, you need to formulate the property as a relational type and manually prove that the program has that type. This has to be done for each program on an individual basis — there are no overarching security properties that hold for all verified programs.

Intransitive noninterference [10] is a declassification mechanism whereby certain specific downward flows are allowed in the label lattice. The system formally verifies that a program obeys the explicitly-allowed flows. These special flows are

intransitive — e.g., we might allow Alice to declassify data to Bob and Bob to declassify to Charlie, but that does not imply that Alice is allowed to declassify to Charlie. The intransitive noninterference system is used to verify simple imperative programs; their language is basically the same as ours, except without the heap-related commands. One idea for future work is to generalize our state predicate P into an action G that precisely describes the transformation that a program is allowed to make on the state. If we implemented this idea, it would be easy to embed the intransitive noninterference system. The action G would specify exactly which special flows are allowed (e.g., the data's label can be changed from Alice to Bob or from Bob to Charlie, but not from Alice to Charlie directly). Ideally, we would have a formal noninterference theorem in terms of G that would give the same result as the formal guarantee in [10].

All of the language-based IFC systems mentioned so far, including our own system, use static reasoning. There are also many dynamic IFC systems (e.g., [2, 7, 18, 20]) that attempt to enforce security of a program during execution. Because dynamic systems are analyzing information flow at runtime, they will incur some overhead cost in execution time. Static IFC systems need not necessarily incur extra costs. Indeed, in our system we have a "true machine" that executes on states with all labels erased. The security-aware machine is for reasoning purposes only; it will never be physically executed.

Sabelfeld and Sands [17] define a road map for analyzing declassification policies in terms of four dimensions: *who* can declassify, *what* can be declassified, *when* can declassification occur, and *where* can it occur. Our notion of declassification can talk about any of these dimensions if we construct the precondition in the right way. The *who* dimension is most naturally handled via the label lattice, but one could also imagine representing principals explicitly in the program state and reasoning about them in the logic. The *what* dimension is handled by default, as the program state contains all of the data to be declassified. The *when* dimension can easily be reasoned about by including a time field in the state. Similarly, the *where* dimension can be reasoned about by including an explicit program counter in the state.

7 Conclusion

In this paper, we described a novel program logic for reasoning about information flow in a low-level language. The primary novelties of our system include:

1. Information flow reasoning (including declassification) in the presence of pointer arithmetic.
2. Connecting the static enforcement mechanism with a dynamic semantics that tracks propagation of security labels.
3. Reasoning about labels conditioned on state predicates. As far as we are aware, the example program of Section 4 (which makes use of conditional labels) cannot be verified as secure in any other IFC system.

In the future, we hope to extend our work to handle termination-sensitivity, dynamic memory allocation/deallocation, nondeterminism, and concurrency.

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, pages 91–102, 2006.
2. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, pages 113–124, 2009.
3. A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.
4. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
5. N. Heintze and J. G. Riecke. The slam calculus: Programming with secrecy and integrity. In *POPL*, pages 365–377, 1998.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
7. C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *IEEE Symposium on Security and Privacy*, pages 3–17, 2013.
8. M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP*, pages 321–334, 2007.
9. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.
10. H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, pages 129–145, 2004.
11. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, 1997.
12. A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179, 2011.
13. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19, 2001.
14. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
15. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
16. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003.
17. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
18. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Haskell*, pages 95–106, 2011.
19. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
20. J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pages 85–96, 2012.
21. A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.
22. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI*, pages 263–278, 2006.

8 Appendix

$$\frac{[\![E]\!]s = \texttt{Some}\ n}{\langle (s,h),\ x := E\rangle \longrightarrow \langle (s[x \mapsto n], h),\ \texttt{skip}\rangle}\ (\text{ASSGN})$$

$$\frac{[\![E]\!]s = \texttt{Some}\ n_1 \qquad h(n_1) = \texttt{Some}\ n_2}{\langle (s,h),\ x := [E]\rangle \longrightarrow \langle (s[x \mapsto n_2], h),\ \texttt{skip}\rangle}\ (\text{READ})$$

$$\frac{[\![E]\!]s = \texttt{Some}\ n_1 \qquad h(n_1) \neq \texttt{None} \qquad [\![E']\!]s = \texttt{Some}\ n_2}{\langle (s,h),\ [E] := E'\rangle \longrightarrow \langle (s, h[n_1 \mapsto n_2]),\ \texttt{skip}\rangle}\ (\text{WRITE})$$

$$\frac{[\![E]\!]\tau = \texttt{Some}\ n}{\langle \tau,\ \texttt{output}\ E\rangle \xrightarrow{[n]} \langle \tau,\ \texttt{skip}\rangle}\ (\text{OUTPUT})$$

$$\frac{[\![B]\!]\tau = \texttt{Some true}}{\langle \tau,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\rangle \longrightarrow \langle \tau,\ C_1\rangle}\ (\text{IF-TRUE})$$

$$\frac{[\![B]\!]\tau = \texttt{Some false}}{\langle \tau,\ \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\rangle \longrightarrow \langle \tau,\ C_2\rangle}\ (\text{IF-FALSE})$$

$$\frac{[\![B]\!]\tau = \texttt{Some true}}{\langle \tau,\ \texttt{while}\ B\ \texttt{do}\ C\rangle \longrightarrow \langle \tau,\ C; \texttt{while}\ B\ \texttt{do}\ C\rangle}\ (\text{WHILE-TRUE})$$

$$\frac{[\![B]\!]\tau = \texttt{Some false}}{\langle \tau,\ \texttt{while}\ B\ \texttt{do}\ C\rangle \longrightarrow \langle \tau,\ \texttt{skip}\rangle}\ (\text{WHILE-FALSE})$$

$$\frac{\langle \tau,\ C_1\rangle \xrightarrow{o} \langle \tau',\ C_1'\rangle}{\langle \tau,\ C_1; C_2\rangle \xrightarrow{o} \langle \tau',\ C_1'; C_2\rangle}\ (\text{SEQ}) \qquad \frac{}{\langle \tau,\ \texttt{skip}; C\rangle \longrightarrow \langle \tau,\ C\rangle}\ (\text{SKIP})$$

Fig. 6. Standard Operational Semantics

End-to-End Verification of Stack-Space Bounds for C Programs

Quentin Carbonneaux Jan Hoffmann Tahina Ramananandro Zhong Shao
Yale University
{quentin.carbonneaux, jan.hoffmann, tahina.ramananandro, zhong.shao}@yale.edu

Abstract

Verified compilers guarantee the preservation of semantic properties and thus enable formal verification of programs at the source level. However, important quantitative properties such as memory and time usage still have to be verified at the machine level where interactive proofs tend to be more tedious and automation is more challenging.

This article describes a framework that enables the formal verification of stack-space bounds of compiled machine code at the C level. It consists of a verified CompCert-based compiler that preserves quantitative properties, a verified quantitative program logic for interactive stack-bound development, and a verified stack analyzer that automatically derives stack bounds during compilation.

The framework is based on event traces that record function calls and returns. The source language is CompCert Clight and the target language is x86 assembly. The compiler is implemented in the Coq Proof Assistant and it is proved that crucial properties of event traces are preserved during compilation. A novel quantitative Hoare logic is developed to verify stack-space bounds at the CompCert Clight level. The quantitative logic is implemented in Coq and proved sound with respect to event traces generated by the small-step semantics of CompCert Clight. Stack-space bounds can be proved at the source level without taking into account low-level details that depend on the implementation of the compiler. The compiler fills in these low-level details during compilation and generates a concrete stack-space bound that applies to the produced machine code. The verified stack analyzer is guaranteed to automatically derive bounds for code with non-recursive functions. It generates a derivation in the quantitative logic to ensure soundness as well as interoperability with interactively developed stack bounds.

In an experimental evaluation, the developed framework is used to obtain verified stack-space bounds for micro benchmarks as well as real system code. The examples include the verified operating-system kernel CertiKOS, parts of the MiBench embedded benchmark suite, and programs from the CompCert benchmarks. The derived bounds are close to the measured stack-space usage of executions of the compiled programs on a Linux x86 system.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Processors—Compilers

Keywords Formal Verification, Compiler Construction, Program Logics, Stack-Space Bounds, Quantitative Verification

PLDI '14, June 9 - 11, 2014, Edinburgh, United Kingdom.

ACM 978-1-4503-2784-8/14/06. . . $15.00.
http://dx.doi.org/10.1145/2594291.2594301

1. Introduction

It has been shown that formal verification can greatly improve software quality [25, 33, 35]. Consequently, formal verification is extensively studied in ongoing research and there exist sophisticated tools that can verify important program properties automatically. However, the most interesting program properties are undecidable and user interaction is therefore inevitable in formal verification.

If a software system is (partly or entirely) developed in a high-level language then the question arises on which language level the verification should be carried out. Formal verification at the source level has the advantage that a developer can interact with the verification tools using the code she has developed. This is beneficial because the compiled code can substantially differ from the source code and low-level code is harder to understand. Moreover, even fully automatic tools profit from the control-flow information and the structure that is available at higher abstraction layers. The disadvantage of verification at the source level is that tools such as compilers have to be part of the trusted computing base and that the verified properties are not directly guaranteed for the code that is executed on the system.

Formally verified compilers [11, 24] such as the CompCert C Compiler [27] guarantee that certain program properties of the source programs are preserved during compilation. As a result, CompCert enables source-level verification of the preserved properties of the compiled code without increasing the size of the trusted computing base.[1] In fact, this has been one of the main motivations for the development of CompCert [27]. However, important quantitative properties such as memory and time consumption are not modeled nor preserved by CompCert and other verified compilers [11, 24]. Such quantitative properties are nevertheless crucial in the verification of safety-critical embedded systems. For example, the DO-178C standard, which is used by in the avionics industry and by regulatory authorities, requires verification activities to show that a program in executable form complies with its requirements on stack usage and worst-case execution time (WCET) [28].

Quantitative program requirements such as stack usage and WCET are usually directly checked at the machine or assembly-code level "since only at this level is all necessary information available" [34]. For stack-space bounds there exist commercial abstract interpretation–based tools—such as Absint's *StackAnalyzer* [14]—that operate directly on machine code. While such tools can derive many simple bounds automatically, they rely on user annotations in the machine code to obtain bounds for more involved programs. The produced bounds are usually not parametric in the input, and the analysis is not modular and only applies to specific hardware platforms. Additionally, the used analysis tools rely on the correctness of the user annotations and are not formally verified.

In this article, we present the first framework for deriving formally verified end-to-end stack-space bounds for C programs. Stack bounds are particularly interesting because stack overflow

[1] If we assume that all verification is carried out using the same trusted base.

is "one of the toughest (and unfortunately common) problems in embedded systems" [13]. Moreover, stack-memory is the only dynamically allocated memory in many embedded systems and the stack usage depends on the implementation of the compiler. While we focus exclusively on stack bounds in this article, our framework is developed with other quantitative resources in mind. Many of the developed techniques can be applied to derive bounds for resources such as heap memory or clock cycles. However, for clock-cycle bounds there is a lot of additional work to be done that is beyond the scope of this article (e.g., developing a formal model for hardware caches and instruction pipelines).

The main innovation of our framework is that it enables the formal verification of stack bounds for compiled x86 assembly code at the C level. To gain the benefits of source-level verification without the entailed disadvantages, we have to deal with three main challenges.

1. We have to model the stack consumption of programs at the C level and we have to formally prove that our model is consistent with the stack consumption of the compiled code.
2. We have to design and implement a C-level verification mechanism that allows users to derive parametric stack-usage bounds in an interactive and flexible way.
3. We have to minimize user interaction during the verification to enable the verification of large systems.

To meet Challenge 1, we use event traces and verified compilation. Our starting point is the CompCert C Compiler. It relies on event traces to prove that a compiled program is a refinement of the source program. We extend event traces with events for function calls and returns and define a *weight* for event traces. The weight describes the stack-space consumption of one program execution as a function of a cost metric that assigns a cost to individual call and return events. The idea is that a user or an (semi) automatic analysis tool derives bounds on the weights of event traces that depend on the stack-frame sizes of the program functions. During compilation the compiler produces a specific cost metric that guarantees that the weight of an event trace under this metric is an upper bound on the stack-space usage of the compiled assembly program which produces this trace. As a result, we derive a verified upper bound if we instantiate the derived memory bound with the cost metric produced by the compiler.

We implemented the extended event traces for full CompCert C and all intermediate languages down to x86 assembly in Coq. We extended CompCert's soundness theorem to take into account the weights of traces. In addition to CompCert's refinement theorem for the original event traces, we prove that compiled programs produce extended event traces whose weights are less than or equal to the weights of the traces at the source level. This means that we allow reordering or deletion of call and return events as long as the weight of the trace is reduced or unchanged. To relate the weight of traces to the execution on a system with finite stack space, we modified the CompCert x86 assembly semantics into a more realistic x86 assembly that features a finite stack, and reimplemented the assembly generation pass of CompCert to our new x86 assembly semantics.

To meet Challenge 2, we have developed and implemented a novel quantitative Hoare logic for CompCert Clight in Coq. To account for memory consumption, the assertions of the logic generalize the usual boolean-valued assertions of Hoare logic. Instead of the classic *true*, our quantitative assertions return a natural number that indicates the amount of memory that is needed to execute the program. The boolean *false* is represented by ∞ and indicates that there are no guarantees provided for the future execution.

We proved the soundness of our quantitative Hoare logic with respect to Clight and CompCert's continuation-based small-step semantics. The soundness theorem states that Hoare triples that are derived with our inference rules describe sound bounds on the weights of traces. The logic can be used for interactive stack-bound development or as a backend for verified static analysis tools. For clarity, we do not prove the safety of programs and simply assume that this is done using a different tool such as Appel's separation logic for Clight [3]. It would be possible to integrate our logic into a separation logic for safety proofs. This would however diminish the deployability of the quantitative logic as a backend for static stack-bound analysis tools since they would be required to also prove memory safety.

To meet Challenge 3, we implemented an automatic stack analyzer for C programs. To verify the soundness of the stack analyzer each successful run generates a derivation in the quantitative Hoare logic. Not only does this simplify the verification, but it also allows interoperability with stack bounds that have been interactively developed in the logic or derived by some other static analysis. Conceptually, our stack analyzer is rather simple but we have proved that it derives bounds for all programs without recursion and function pointers. This is already sufficient for many programs that are used in embedded systems. Using our automatic analysis we have created a verified C compiler that translates a program without function pointers and recursive calls to x86 assembly and automatically derives a stack bound for each function in the program including $\mathsf{main}()$.

We have successfully used our framework to verify end-to-end memory bounds for micro benchmarks and system software. Our main example is the CertiKOS [15] operating system kernel that is currently under development at Yale. Our automatic analyzer finds stack bounds for all functions in the simplified development version of CertiKOS that is currently verified. Other examples are taken from Leroy's CompCert benchmarks and the MiBench embedded benchmark suite [17]. To evaluate the quality of the verified stack-space bounds, we experimentally compared the automatically and manually verified bounds with the actual stack-space consumption during the execution of the compiled C programs. Our experiments indicate that both the manually and automatically derived bounds over-approximate the stack usage by exactly four bytes. More details can be found in Section 6.

In summary, we make the following contributions.

- We introduce a methodology that uses cost metrics to link event traces to resource consumption. This approach enables us to link source-level code to the resource consumption of compiled target-level code.
- We develop a novel quantitative Hoare logic to reason about the resource consumption of programs at the source level. We have formally verified the soundness of the logic with respect to CompCert Clight in Coq.
- We introduce *Quantitative CompCert*, a modified version of the verified CompCert C Compiler, in which parametric stack bounds are preserved during compilation. Furthermore, Quantitative CompCert creates a cost metric so that the instantiation of the bounds with the metric forms an upper bound on the memory consumption of the compiled code.
- We have implemented and verified an automatic stack analyzer.
- We have evaluated the practicability of our framework with experiments using micro benchmarks and system code.

The complete Coq development and the implemented tools are well documented and publically available on the authors' websites. The *PLDI Artifact Evaluation Committee* reproduced samples of our experiments and tested the implemented tools on additional programs. The reviewers unanimously stated that our implementation *exceeded their expectations.* A companion technical report [9] contains additional explanation, lemmas, and examples.

2. An Illustrative Example

In this section, we sketch the verification of stack-space bounds for an example program in our framework. Figure 1 shows a C program with two integer parameters: ALEN and SEED.

This program will fill an array of size ALEN with an increasing sequence of pseudo random integers and search through it. The random numbers are created by a linear congruential generator initialized by the SEED parameter. The search procedure used is a binary search implemented in the recursive function search.

Our goal is to derive stack bounds for the compiled x86 assembly code of the program that are verified with respect to our accurate x86 model in Coq. The first step is to create an abstract syntax tree of the code in Coq. This can be done automatically, for instance by using CompCert's parsing mechanism. The second step is to use our quantitative Hoare logic to prove bounds on the function calls that are performed when executing main.

To relate function calls and returns at different abstraction levels during compilation we use call and return events. For instance, an execution of main could produce the following trace.

$$\mathsf{call}(\mathsf{main}), \mathsf{call}(\mathsf{init}), \mathsf{call}(\mathsf{random}), \mathsf{ret}(\mathsf{random}), \mathsf{ret}(\mathsf{init}),$$
$$\mathsf{call}(\mathsf{search}), \mathsf{call}(\mathsf{search}), \mathsf{ret}(\mathsf{search}), \mathsf{ret}(\mathsf{search}), \mathsf{ret}(\mathsf{main})$$

From such a trace and a metric M that maps each function name in the program to its stack-frame size, we can obtain the stack usage of the execution that produced the trace. For the previous example trace, we can for instance derive the following stack usage.

$$M(\mathsf{main}) + \max\{M(\mathsf{init}) + M(\mathsf{random}), 2 \cdot M(\mathsf{search})\}$$

In classical Hoare logic, assertions map program states to Booleans. In our quantitative Hoare logic assertions map program states to non-negative numbers. Intuitively, the meaning of a quantitative Hoare triple $\{P\}\, S\, \{Q\}$ is the following. For every program state σ, $P(\sigma)$ is an upper bound on the stack consumption of the statement S started in state σ. Furthermore, Q describes the stack space that has become available after the execution, as a function of the final program state. This is similar to type systems and program logics for amortized resource analysis [5, 21].

We implemented a function in Coq that automatically computes a derivation in the quantitative logic for a program without recursive functions. Using this automatic stack analyzer, we derive for instance the following triple for the function call init().

$$\{M(\mathsf{init}) + M(\mathsf{random})\}\, \mathsf{init}()\, \{M(\mathsf{init}) + M(\mathsf{random})\}$$

For functions making use of recursion such as search, we derive a quantitative triple interactively using Coq. For search we derive

$$\{\mathrm{L}(\mathsf{end} - \mathsf{beg})\}\, \mathsf{search}(\mathsf{elem}, \mathsf{beg}, \mathsf{end})\, \{\mathrm{L}(\mathsf{end} - \mathsf{beg})\}$$

where $L(\Delta) = M(\mathsf{search}) \cdot (2 + \log_2(\Delta))$. Since the mathematical $\log_2$ function is undefined on non-positive values, we take as convention that $\log_2(\Delta) = +\infty$ when $\Delta < 0$ and $\log_2(0) = 0$. This trick allows us to simulate a logical precondition stating that beg must be lower or equal to end before calling search.

For main we combine the previous results and derive the bound

$$\{M(\mathsf{main}) + N\}\, \mathsf{main}()\, \{M(\mathsf{main}) + N\}$$

where $N = \max(M(\mathsf{init}) + M(\mathsf{random}), \mathrm{L}(\mathsf{ALEN}))$. To be able to derive this bound on the main function we have to require that $0 < \mathsf{ALEN} \leqslant 2^{32} - 1$, in the Coq development this is stated as a section hypothesis which will later be instantiated when ALEN is chosen by the user before compiling.

The third and final step in the derivation of the stack bounds is to compile the program with Quantitative CompCert, our modified CompCert C Compiler. The compiler produces x86 assembly code and a concrete metric M_0. It follows from CompCert's correctness theorem that the compiled code is a semantic refinement of our source program. In addition, we have formally verified that the metric M_0 correctly relates the abstractly defined stack consumption—using the event traces—to the actual stack consumption in our abstract x86 machine. Moreover, we have verified that applying M_0 to the preconditions in the triples of the quantitative Hoare logic results in sound stack bounds on the x86 machine. The final bounds that we obtain for our examples are for instance 32 bytes for init() and $112 + 40 \cdot \log_2(\mathsf{ALEN})$ bytes for main().

```
typedef unsigned int u32;
u32 a[ALEN];
u32 seed = SEED;

u32 search(u32 elem, u32 beg, u32 end) {
       u32 mid = beg + (end-beg) / 2;
       if (end-beg <= 1) return beg;
       if (a[mid] > elem) end = mid;
       else beg = mid;

       return search(elem, beg, end);
}
u32 random() {
       seed = (seed * 1664525) + 1013904223;
       return seed;
}
void init() {
       u32 i, rnd, prev = 0;

       for (i=0; i<ALEN; i++) {
              rnd = random();
              a[i] = prev + rnd % 17;
              prev = a[i]; }
}
int main() {
       u32 idx, elem;
       init();
       elem = random() % (17 * ALEN);
       idx = search(elem, 0, ALEN);
       return a[idx] == elem;
}
```

Figure 1. An illustrative example for static stack-bound computation. Constant stack bounds for the non-recursive functions are derived automatically. The logarithmic bound for the function search is derived with a hand-crafted proof in our quantitative Hoare logic.

3. Quantitative CompCert: Verified Stack-Aware Compilation

In this section, we introduce our new technique for verifying *quantitative compiler correctness* and its implementation in Quantitative CompCert. We focus on stack-space usage but believe that similar techniques can be used to bound the time and heap-space requirements of programs. Our development is highly influenced by the design of CompCert [27], a verified compiler for the C language. CompCert C accepts most of the ISO-C-90 language and produces machine code for the IA32 architecture (among others). CompCert uses 11 intermediate languages and 20 passes to compile a C AST to x86 assembly.

The soundness proof of CompCert is based on trace-based operational semantics for the source, target, and intermediate languages. These semantics generate traces of events during the execution of programs. Events include input/output and external function calls. The soundness theorem of CompCert states that every event trace that can be generated by the compiled program can also be generated by the source program provided that the source program does not go wrong. In other words, the compiled program is a refinement of the source program with respect to the observable events.

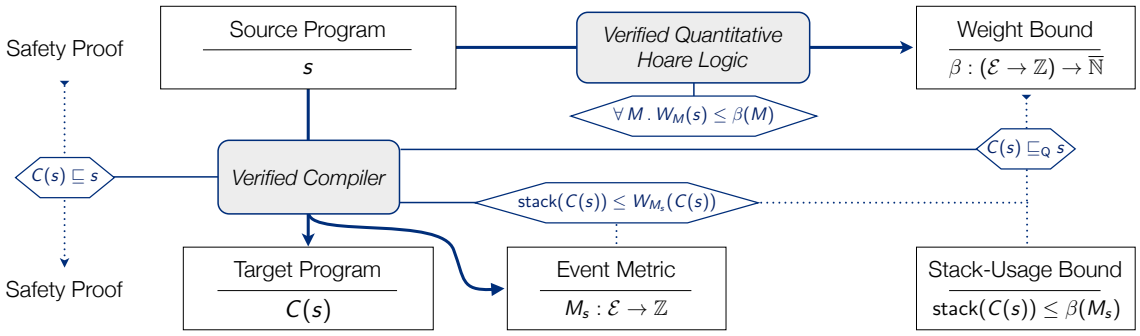


Figure 2. Overview of our quantitative verification framework. We write $W_M(s) = \sup\{W_M(B) \mid B \in [\![s]\!]\}$ for the weight of the program s under the metric M. We write $\mathsf{stack}(s)$ for the smallest number n so that s runs without stack overflow if executed with a stack of size n.

3.1 Quantitative Compiler Correctness

In the following, we show how to extend trace-based compiler-correctness proofs to also cover stack-space consumption. In short, our technique works as follows.

1. We generate events for all semantic actions that are relevant for stack-space usage, that is, function calls and returns.
2. We define a *weight* function for event traces that describes the stack-space consumption of program executions that produce that trace. The weight of an event trace is parameterized by a resource metric that describes the cost of each event.
3. We formally verify that for all resource metrics and for all event traces produced by a target program, the source program either goes wrong or produces an equivalent (see the following definition) event trace with a greater or equal weight.
4. During compilation, we produce a cost metric that accurately describes the memory consumption of target programs: If an execution of a target program produces an event trace of weight n under the produced metric then this execution can be performed on a system with stack size n.

We now formalize and elaborate on these points.

Event Traces In CompCert, the observable events are external function calls (e.g., I/O events) that are represented by function identifiers together with a list of input values and an output value as given by the following grammar. To track stack usage, we add memory events for internal function calls and returns. In contrast to I/O events, memory events do not have to be preserved during compilation.

$$\begin{aligned} \text{Event values} \quad & v ::= \mathsf{int}(n) \mid \mathsf{float}(q) \\ \text{I/O events} \quad & \nu ::= f(\vec{v} \mapsto v) \\ \text{Memory events} \quad & \mu ::= \mathsf{call}(x) \mid \mathsf{ret}(x) \end{aligned}$$

Event traces are defined in a similar way to CompCert. We distinguish finite (inductive) traces t and possibly infinite (coinductive) traces T. A program behavior is either a converging computation $\mathsf{conv}(t, n)$ producing a finite event trace t and a return code n, a diverging computation $\mathsf{div}(T)$ producing a finite or infinite trace T, or a computation $\mathsf{fail}(t)$ that goes wrong and produces the finite trace t.

$$\begin{aligned} \text{Finite event traces} \quad & t ::= \epsilon \mid \nu \cdot t \mid \mu \cdot t \\ \text{Coinductive event traces} \quad & T ::= \epsilon \mid \nu \cdot T \mid \mu \cdot T \\ \text{Behaviors} \quad & B ::= \mathsf{conv}(t, n) \mid \mathsf{div}(T) \mid \mathsf{fail}(t) \end{aligned}$$

We write $\mathcal{E}$ for the set of memory and I/O events, $\mathcal{B}$ for the set of behaviors, and $\mathcal{T}$ for the set of traces.

Weights of Behaviors For a behavior B, we define the set of finite prefix traces $\mathit{prefs}(B)$ of B as follows.

$$\begin{aligned} \mathit{prefs}(\mathsf{conv}(t, n)) &= \{t_1 \mid t = t_1 \cdot t_2\} \\ \mathit{prefs}(\mathsf{div}(T)) &= \{t \mid T = t \cdot T'\} \\ \mathit{prefs}(\mathsf{fail}(t)) &= \{t_1 \mid t = t_1 \cdot t_2\} \end{aligned}$$

The *weight* $W_M(B) \in \mathbb{N} \cup \{\infty\}$ of a behavior B describes the number of bytes that are needed in an execution that produces B. It is parameterized by a resource metric $M : \mathcal{E} \to \mathbb{Z}$ that maps events to integers (bytes). The purpose of the metric in our work is to relate memory events to the sizes of the stack frames of functions in the target code. To this end, we only use stack metrics, that is, metrics M such that for all functions f and for all external functions g

$$0 \leqslant M(\mathsf{call}(f)) = -M(\mathsf{ret}(f)) \quad \text{and} \quad M(g(\vec{v} \mapsto v)) = 0\,.$$

In the Coq implementation of our compiler, we can also deal with nonzero stack consumption for external functions as long as the stack consumption of each call is bounded by a constant.

Before we define the weight, we first inductively define the valuation $V_M(t)$ of a finite trace t.

$$V_M(\epsilon) = 0 \qquad \text{and} \qquad V_M(\alpha \cdot t) = V_M(t) + M(\alpha)$$

We now define the weight $W_M(T)$ of a potentially infinite trace T and the weight $W_M(B)$ of a behavior B under the metric M as follows:

$$\begin{aligned} W_M(T) &= \sup\{V_M(t) \mid T = t \cdot T'\} \\ W_M(B) &= \sup\{V_M(t) \mid t \in \mathit{prefs}(B)\} \end{aligned}$$

Quantitative Refinement For our description of quantitative refinements we leave the definition of programs abstract. A program $s \in \mathcal{P}$ is simply an object that is associated, through a function $[\![\cdot]\!] : \mathcal{P} \to \mathcal{B}$, with a set of behaviors $[\![s]\!] \in \mathcal{B}$. An execution of a program can produce different traces, either due to non-determinism in the semantics or due to user inputs recorded in the event traces.

For a behavior B we define the pruned behavior as the behavior $\overline{B}$ that results from deleting all memory events ($\mathsf{call}(x)$ or $\mathsf{ret}(x)$) from B. The formal definition can be found in the TR [9].

In CompCert, compiler correctness is formalized through the notion of *refinement*. A (target) program s' is a refinement of a (source) program s, written $s' \prec s$, if for every behavior $B' \in [\![s']\!]$ there is $B \in [\![s]\!]$ such that $\overline{B} = \overline{B'}$ or $\mathsf{fail}(t) \in [\![s]\!]$ for some trace t.[2] Note that memory events are not taken into account in CompCert's classic definition of refinement.

[2] In fact, it is enough to prove that $\overline{B'} \sim \overline{B}$ (bisimilarity of infinite traces), because $[\![s]\!]$ is closed by bisimilarity.

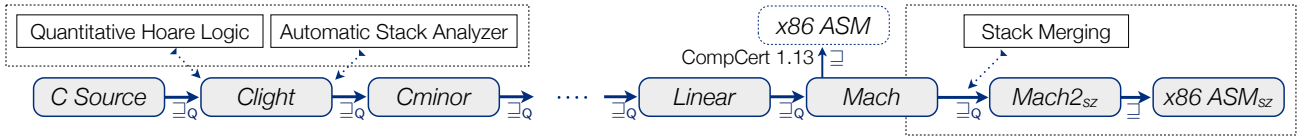


Figure 3. Our modified stack-aware CompCert C compiler. We replace CompCert's x86 assembly with the more realistic x86 assembly semantics ASM_{sz} with finite stack. Pseudo assembly instructions such as Pallocframe and Pfreeframe are not needed anymore.

To also relate the memory events in the behaviors of two programs, we define a novel *quantitative refinement*. A (target) program s' is a quantitative refinement of a (source) program s, written $s' \prec_Q s$ if the following holds. For every behavior $B' \in [\![s']\!]$ there exists $B \in [\![s]\!]$ such that $\overline{B} = \overline{B'}$ and $W_M(B') \leqslant W_M(B)$ for all stack metrics M, or $\mathsf{fail}(t) \in [\![s]\!]$ for some trace t. In Quantitative CompCert, our modified CompCert compiler, we prove for each compiler pass C that $C(s) \prec_Q s$ for every program s.

Verifying Stack-Space Usage Figure 2 summarizes how we verify the stack-space usage of a program in our framework. First, we prove a bound $\beta : (\mathcal{E} \to \mathbb{Z}) \to \mathbb{N}$ on the weights of the event traces that a program can produce. This bound is parameterized by an event metric $M : \mathcal{E} \to \mathbb{Z}$. Second, our verified compiler—thanks to quantitative refinement—ensures that the computed bound also holds for the weights of the traces of the compiled program.

Third, we have to relate the computed bound to the actual stack usage of the compiled code. Therefore, our compiler computes not only a target program $C(s)$ but also a metric M_s such that $C(s)$ can be safely executed with a stack-memory size of $\sup\{W_{M_s}(B) \mid B \in [\![C(s)]\!]\}$ bytes. As a result, the initially derived bound for the source code can be instantiated with the metric M_s to obtain the wanted stack-space bound $M_s(\beta)$ for the target program.

In this overview picture, we assume that the semantics of the target and source languages are both formulated with an unbounded stack. The final step of the soundness proof (not illustrated in Figure 2) is to relate the trace-based semantics of the target language to a realistic assembly semantics in which the program is executed with a fixed stack size. To this end, we prove that an execution of $C(s)$ with bounded stack space $\sup\{W_{M_s}(B) \mid B \in [\![C(s)]\!]\}$ is a refinement of the execution of $C(s)$ in the semantics with unbounded stack (see explanation in Section 3.2).

3.2 Verification and Implementation

We implemented the verification framework that we outlined in Section 3.1 for the CompCert C compiler using the proof assistant Coq. The verification consists of about 5000 lines of Coq code that we integrated into CompCert 1.13 (which originally consists of about 90000 lines of Coq code) to obtain a modified version that we call *Quantitative CompCert*. CompCert 1.13 is decomposed into 20 passes between 11 intermediate languages (see [9] for an overview). We describe our modified Quantitative CompCert in this section.

The problem: stack consumption in CompCert For each intermediate language of CompCert (beyond C subsets), each function call allocates a memory region—called the *stack frame*—to store its addressable local variables, and later the spilling locations and the function arguments to handle the calling conventions. This stack frame is freed upon function return. However, even though each stack frame is finite, there may well be an unbounded number of such allocations, even for nested function calls. Indeed, in CompCert, allocating a stack frame always succeeds, thus CompCert does not model *stack overflow*.

Our solution: Quantitative CompCert In Quantitative CompCert, we overcome this issue by modifying the semantics of the target assembly language. We preallocate a finite memory region for the whole stack, into which all stack frames shall be merged together during the execution instead of being individually allocated.

By contrast, we still want the source and intermediate languages to allocate an individual stack frame per function call. First, we want to change CompCert only if necessary so as to still support all features of CompCert C. Second, it would not be very meaningful to introduce a finite stack at a high language level since it is unclear how to model stack sizes. The only major change we bring to those languages is to introduce our call and return events into the trace.

As shown in Figure 3, this leads us to split CompCert into two parts. In the first part, we compile CompCert C down to the CompCert Mach low-level language (which comes just before assembly generation) by adapting the proofs of existing passes to quantitative refinement. In the second part, we perform two passes to merge all stack frames together. The key point of our work is that this second part will require the Mach traces to not stack overflow, which justifies the use of quantitative refinement for the first part.

Quantitative Refinement In the first part of the compiler, from CompCert C down to Mach, we add call and return events to the semantics of each language, at the level of each function call and return (as described in Section 3.1). This change is uniform in all languages between CompCert C to Mach: indeed, in each small-step operational semantics, the rules responsible for internal function call and return all have the same shape.

Then, thanks to these changes, we support all of CompCert 1.13 passes except two optional optimizations (see Section 3.3), and, with no significant changes to the proofs, we prove that they exactly preserve traces with function call events.

Generation of Target Cost Metric The semantics of CompCert C allocates a separate memory region for each addressable local variable. In Mach, all those variables as well as the spilling locations, the function arguments, and the return address are stored in a stack frame. Actually, the stack frame of a Mach function call is completely laid out, so that no additional memory is necessary when generating the CompCert x86 assembly code. This means that, at the level of Mach, we already know the stack size necessary for a function call (thanks to the fact that the original CompCert does not support some C features, see 3.3): for a given function, this size is constant and does not depend on the arguments nor the input. So, we can use the sizes of Mach stack frames as cost metric for functions to accurately estimate stack bounds at the source level.

Generation of Assembly Code Recall that CompCert x86 assembly language is not realistic enough as it does not prevent a program from allocating an infinite number of stack frames. Our goal, as one of our main applications of our quantitative refinement, is to make the CompCert x86 assembly language more realistic by having it model a contiguous finite stack that is preallocated at the beginning of the program. The semantics of our new CompCert x86 assembly is parameterized by the size $sz + 4$ [3] of the whole stack (provided, in most cases, by the host operating system). We call this new x86 semantics ASM_{sz}. We design it in such a way that an execution goes

[3] sz is the stack size actually consumed by the program starting from `main`, but we have to account for the return address of the "caller" of `main`

wrong if the program tries to access more than sz bytes of stack. In other words, stack overflow becomes possible in ASM_{sz}.

Because the notion of function call is no longer relevant (there is no "control stack"), we lose the ability to extend this semantics with call and return events. So, rather than quantitative refinement, we are actually interested in whether a CompCert C source program can run on ASM_{sz} without going wrong because of stack overflow. The correctness of our Quantitative CompCert compiler is formalized by the following theorem.

Theorem 1. *Let $sz + 4 \in [4, 2^{32})$ be the size of the whole target stack. Consider a CompCert C source program S and assume the following:*

1. *S does not go wrong in the ordinary setting of unbounded stack space, that is, $\nexists t, \mathsf{fail}(t) \in [\![S]\!]$.*
2. *Quantitative CompCert produces a Mach intermediate target code I, with the sizes of stack frames[4] SF and the subsequent cost metric $M(f) = \mathsf{SF}(f) + 4$.*
3. *The stack bounds of S inferred at the source level are lower than sz under the Match cost metric M: $\forall B \in [\![S]\!], W_M(B) \leqslant sz$.*
4. *From I, our compiler produces a target assembly code T.*

Then, when run in ASM_{sz}, T refines S in the sense of CompCert: $\forall B' \in [\![T]\!]_{sz}, \exists B \in [\![S]\!], B' = B$. In particular, T cannot go wrong and thus does not stack overflow.

It is important to first prove that S cannot go wrong in unbounded stack space. Indeed, the correctness of our assembly generation depends on the fact that the weights of Mach traces are lower than sz. If S were to have a wrong behavior $\mathsf{fail}(t)$ then I might actually have a behavior $t \cdot B$ whose weight could well exceed sz even though $W_M(\mathsf{fail}(t))$ does not. As each pass is proved independently of the others, it is not possible to track the behaviors of I that could potentially come from wrong behaviors of S.

In the original CompCert x86 assembly language, the notion of stack frame is still kept, so that this language has two *pseudo-instructions* Pallocframe and Pfreeframe responsible of allocating and freeing the corresponding memory block, even though those pseudo-instructions are then turned into real x86 assembly instructions performing pointer arithmetics with the ESP stack pointer register. This latter transformation cannot be proved correct in CompCert because pointer arithmetics cannot cross block boundaries in the CompCert memory model. Therefore this transformation is done in an unverified "pretty-printing" stage, after CompCert has generated the x86 assembly code of the source program.

Our new assembly semantics overcomes this limitation. Now, instead of allocating different memory blocks, we preallocate one single block of size $sz + 4$ at the beginning of the program to hold the whole stack, and our assembly generation pass ensures that the value of ESP always points within this block. Therefore the pseudo-instructions are no longer necessary, and the pointer arithmetics needed at function entry and exit can be performed within our formalized ASM_{sz} assembly language.

As an interesting side effect, accessing function arguments is now simpler in our assembly language. Indeed, in the x86 calling convention, a function has to look for its arguments in the stack frame of its caller. To this purpose, the original CompCert keeps a back pointer to link each stack frame to its parent. Thanks to our changes, function arguments can now be provably accessed through pointer arithmetics with no indirection, so that this back link is no longer necessary. Because in the original CompCert, stack frames are independent memory blocks, it was necessary for the callee to have a pointer to the caller stack frame, called the *back link*, in its own stack frame. The callee could then access its arguments by one indirection through this back link. In our new ASM_{sz} assembly language, stack frames are no longer independent, so that the callee can access its arguments directly by pointer arithmetics within the whole stack block.

[4] In CompCert Mach, the syntax of a program p includes a finite map SF such that, for any function f defined in p, the operational semantics of Mach allocates a stack frame of $\mathsf{SF}(f)$ bytes whenever f is entered.

3.3 Limitations

Neither the original CompCert nor Quantitative CompCert do support variable stack-frame size: C features such as variable-length arrays or dynamic stack allocation (`alloca` special library functions) are not supported. Thus, the size of the stack frame of a Mach function can be computed statically, and can be used to define the cost metric of the program. Moreover, the subsequently produced assembly code does not need to use `push` or `pop`, so any change to the stack pointer is done only through pointer arithmetics.

Quantitative CompCert currently does not support the following optional two optimization passes (that are present in the original CompCert): tail-call recognition and function inlining. We describe how to deal with these two optimizations in the companion TR [9] and the implementation is underway.

4. Quantitative Hoare Logic for CompCert Clight

In this section, we describe the novel quantitative program logic for CompCert Clight. The logic has been formalized and proved sound using Coq. At some points, we simplify the presented logic in comparison to the implemented version to discuss general ideas instead of technical details.

Some particularities of the logic can be better understood with respect to Clight and the continuation-based small-step semantics for Clight programs that is used in CompCert.

4.1 CompCert Clight

CompCert Clight is the most abstract intermediate language used by CompCert. Mainly, it is a subset of C in which loops can only be exited with a break statement and expressions are free of side effects. Using Clight instead of C simplifies the definition of our quantitative program logic and is also in line with the design of CompCert and the verification of CertiKOS.

Syntax We use Clight expressions in the logic. Our statements' syntax is a subset of Clight's to focus on the main ideas of our program logic. For simplicity, loops are infinite unless they are terminated using a break statement. We do not consider function pointers, goto statements, continue statements, and switch statements (see Section 4.4).

$$S, S_1, S_2 ::= \mathsf{skip} \mid x = E \mid x = f(E^*) \mid S_1; S_2 \mid \mathsf{loop}\, S \\ \mid \mathsf{if}\,(E)\,\mathsf{then}\, S_1\, \mathsf{else}\, S_2 \mid \mathsf{break} \mid \mathsf{return}\, E$$

Like in C, a program consists of a list of global variable declarations, a list of function declarations, and the identifier of the main statement, which is the entry point of the program.

4.2 Operational Semantics

CompCert Clight's semantics is based on small-step transitions and continuations. Expressions—which do not have side effects—are evaluated in a big-step fashion. We use a simplified version of Clight's semantics that is sufficient for our subset. It is easy to relate evaluations in our simplified version to evaluations in the original semantics and we have implemented a verified compiler from our simple Clight to Clight with CompCert's original semantics.

Values and Memory Model A value is either an integer n or a memory address ℓ.

$$Val ::= \mathsf{int}\ n \mid \mathsf{adr}\ \ell$$

In the Coq development we use CompCert's memory model. However, the main ideas of the logic can be described with a simple

$$\Gamma \vdash \{Q^s\}\,\mathsf{skip}\,\{Q\}\ (\text{Q:SKIP}) \qquad \Gamma \vdash \{Q^b\}\,\mathsf{break}\,\{Q\}\ (\text{Q:BREAK}) \qquad \Gamma \vdash \{\lambda\sigma\,.\,Q^r\,[\![E]\!]^{\Delta}_{\sigma})\}\,\mathsf{return}\,E\,\{Q\}\ (\text{Q:RETURN})$$

$$\frac{\Gamma \vdash \{I^s\}\,S\,\{I\}}{\Gamma \vdash \{I^s\}\,\mathsf{loop}\,S\,\{(I^b, \bot, I^r)\}}(\text{Q:LOOP}) \qquad \frac{\Gamma(f)=(P_f,Q_f) \quad P=\lambda(\theta,H).P_f([\![E]\!]^{\Delta}_{(\theta,H)},H) \quad Q=\lambda(\theta,H).Q_f([\![x]\!]^{\Delta}_{(\theta,H)},H)}{\Gamma \vdash \{P+M(f)\}\,x=f(E)\,\{(Q+M(f),\bot,\bot)\}}(\text{Q:CALL})$$

$$\frac{\Gamma \vdash \{P\}\,S_1\,\{(R,Q^b,Q^r)\} \quad \Gamma \vdash \{R\}\,S_2\,\{Q\}}{\Gamma \vdash \{P\}\,S_1;S_2\,\{Q\}}(\text{Q:SEQ}) \qquad \frac{c \geqslant 0 \quad \{P\}\,S\,\{Q\}}{\{P+c\}\,S\,\{Q+c\}}(\text{Q:FRAME}) \qquad \frac{P \geqslant P' \quad \{P'\}\,S\,\{Q'\} \quad Q' \geqslant Q}{\{P\}\,S\,\{Q\}}(\text{Q:CONSEQ})$$

Figure 4. Selected rules of the quantitative program logic.

memory model in which locations are mapped to values and labels.

$$H : Mem = \text{Loc} \to Val \cup \{\blacksquare\}$$

The label $\blacksquare$ is used to indicate that a location has been freed and can no longer be used.

Evaluating Expressions Expressions are evaluated with respect to a memory $H : Mem$ and two environments

$$\theta : VID \to Val \qquad \text{and} \qquad \Delta : VID \to \text{Loc}\ .$$

The local environment θ maps local variables to values and the global environment Δ maps global variables to locations. We assume that always $\text{dom}(\Delta) \cap \text{dom}(\theta) = \varnothing$.

The semantics $[\![E]\!]^{\Delta}_{(\theta,H)} = v$ of an expression E under a global environment Δ, a local environment θ, and a memory H is defined by induction on the structure of E.

Continuations The small-step transition relation for statements is based on continuations. Continuations handle the local control flow within a function as well as the logical call stack.

$$K ::= \mathsf{Kstop} \mid \mathsf{Kseq}\,S\,K \mid \mathsf{Kloop}\,S\,K \mid \mathsf{Kcall}\,x\,f\,\theta\,K$$

A continuation K is either the empty continuation Kstop, a sequence $\mathsf{Kseq}\,S\,K$, a loop $\mathsf{Kloop}\,S\,K$, or a stack frame $\mathsf{Kcall}\,x\,f\,\theta\,K$.

Evaluating Statements Statements are evaluated under a program state $(\theta, H) \in State = (VID \to Val) \times Mem$ and a *global environment*

$$(\Sigma, \Delta) : FID \to ([VID] \times \mathcal{S}) \times (VID \to \text{Loc})$$

that maps internal functions to their definitions—a list of argument names and the function body—and global variables to values.

The small-step evaluation rules are given in the companion TR [9]. They define a transition

$$(\Sigma, \Delta) \vdash (S, K, \sigma) \to_{\{\mu|\nu|\epsilon\}} (S', K', \sigma')$$

where μ is a memory event, ν is an I/O event, ϵ denotes no event, S, S' are statements, K, K' are continuations, and $\sigma, \sigma' \in State$.

From the small-step transition relation we derive the following many-step relation in which t is a finite trace. We write

$$(\Sigma, \Delta) \vdash (S_1, K_1, \sigma_1) \to^n_t (S_{n+1}, K_{n+1}, \sigma_{n+1})$$

if $t = a_1, \ldots, a_n$ and there exists (S_i, K_i, σ_i) such that for all i

$$(\Sigma, \Delta) \vdash (S_i, K_i, \sigma_i) \to_{a_i} (S_{i+1}, K_{i+1}, \sigma_{i+1})\ .$$

For a statement S and a continuation K, we define the weight under the global environment (Σ, Δ), the program state σ, and the metric M as $(\Sigma, \Delta) \vdash W_{(\sigma,M)}(S, K) = \sup\{V_M(t) \mid \exists S', K', \sigma', t, n\,.\,(\Sigma, \Delta) \vdash (S, K, \sigma) \to^n_t (S', K', \sigma')\}$.

4.3 Quantitative Hoare Logic

In the following we describe a simplified version of the quantitative Hoare logic that we use in Coq to prove bounds on the weights of the traces of Clight programs. For a given statement S and a continuation K, our goal is to derive a bound $(\Sigma, \Delta) \vdash P(\sigma, M) \in \mathbb{N}$ such that $(\Sigma, \Delta) \vdash P(\sigma, M) \ \geqslant\ (\Sigma, \Delta) \vdash W_{(\sigma,M)}(S, K)$ for all program states σ and resource metrics M. In the remainder of this section we assume a fixed global environment (Σ, Δ).

We generalize classic Hoare logic to express not only classical boolean-valued assertions but also assertions that talk about the future stack-space usage. Instead of the usual assertions $P : State \to bool$ of Hoare logic we use assertions

$$P : State \to \mathbb{N} \cup \{\infty\}\ .$$

This can be understood as a refinement of boolean assertions where *false* is interpreted by ∞ and *true* is refined by $\mathbb{N}$. We write $Assn$ for $State \to \mathbb{N} \cup \{\infty\}$, and $\bot = (\_ \mapsto \infty)$. In the actual implementation, assertions have the type $State \to \mathbb{N} \to \text{Prop}$. For a given $\sigma \in State$, such an assertion can be seen as a set $B \subseteq \mathbb{N}$ of valid bounds. We do this only to use Coq's support for propositional reasoning. The presentation here is easier to read.

The continuation-based semantics of Clight requires that we distinguish pre- and postconditions in the logic to account for different possible ways to exit a block of code. This approach is standard in Hoare logics and followed for instance in Appel's separation logic for Clight [3]. Our postconditions

$$Q = (Q^s, Q^b, Q^r) : Assn \times Assn \times (Val \to Assn)$$

provide one assertion Q^s for the case in which the block is exited by fall through, one assertion Q^b if the block is exited by a break, and a function Q^r from values to assertions in case the block is exited by a return. The function Q^r takes the return value as argument.

Since we have to deal with (possibly recursive) functions, we also need a function context

$$\Gamma : FID \to ((Val \times Mem) \to \mathbb{N} \cup \{\infty\}) \times ((Val \times Mem) \to \mathbb{N} \cup \{\infty\})$$

that maps function names to their specifications, that is, pre- and postconditions. The precondition depends on the value that is passed to the function by the caller and the memory. The postcondition depends on the return value and the memory. We assume that a function has only one argument in this article. In the Coq implementation, an arbitrary number of function arguments is allowed.

In summary, a quantitative Hoare triple has the form $\Gamma \vdash \{P\}\,S\,\{Q\}$ where Γ is a function context, $P : Assn$ is a precondition, $Q : Assn \times Assn \times (Val \to Assn)$ is a postcondition, and S is a statement.

Intuitively, an assertion can be seen as a *potential function* that maps a program state to a non-negative potential. The potential of the precondition P must be sufficient to cover the cost of the execution of the statement S and the potential Q after the execution of S (as in amortized resource analysis [19]).

Rules Figure 4 shows selected rules of the quantitative logic. We lift the operations $+$ and $\geqslant$ pointwise to assertions $P, Q : Assn$. A constant $c \in \mathbb{N} \cup \{\infty\}$ is sometimes used as the constant assertion $\_ \mapsto c$. We fix an event metric M and a global environment (Σ, Δ).

In the Q:SKIP rule, we do not have to account for any stack consumption. As a result, the precondition can be any (potential)

$$\dfrac{\dfrac{\dfrac{\Gamma(f) = (\lambda(v,H)\,.\,0, \lambda(v,H)\,.\,0)}{\Gamma \vdash \{(m_f)\}\,\mathsf{f}()\,\{(m_f,\bot,\bot)\}}\,(\text{Q:CALL})}{\Gamma \vdash \{(m_f + X_f)\}\,\mathsf{f}()\,\{(m_f + X_f,\bot,\bot)\}}\,(\text{Q:FRAME})}{\Gamma \vdash \{\max(m_f,m_g)\}\,\mathsf{f}()\,\{Q\}}\,(\text{EQ}) \qquad \dfrac{\dfrac{\dfrac{\Gamma(g) = (\lambda(v,H)\,.\,0, \lambda(v,H)\,.\,0)}{\Gamma \vdash \{(m_g)\}\,\mathsf{g}()\,\{(m_g,\bot,\bot)\}}\,(\text{Q:CALL})}{\Gamma \vdash \{(m_g + X_g)\}\,\mathsf{g}()\,\{(m_g + X_g,\bot,\bot)\}}\,(\text{Q:FRAME})}{\Gamma \vdash \{\max(m_f,m_g)\}\,\mathsf{g}()\,\{Q\}}\,(\text{EQ})$$

$$\Gamma \vdash \{\max(m_f,m_g)\}\,\mathsf{f}();\mathsf{g}()\,\{Q\}\;(\text{Q:SEQ})$$

where $M(\mathsf{call}(f)) = m_f$ $\quad M(\mathsf{call}(g)) = m_g$ $\quad Q = (\max(m_f,m_g),\bot,\bot)$ $\quad X_\theta = \max(m_f,m_g) - m_\theta$ for $\theta \in \{f,g\}$

Figure 5. An example derivation of a stack-space bound in the quantitative logic.

function. After the execution, the skip part of the postcondition must be valid on the same (unchanged) program state. So we have to make sure that we do not end up with more potential and simply use the precondition as the skip part of the postcondition. The break and return parts of the postcondition are not reachable and can therefore be arbitrary. The rules Q:BREAK and Q:RETURN are similar.

In the Q:SEQ rule we have to account for early exits in statements. For instance, if S_1 contains a break statement then S_2 will never be executed so we must ensure in the break part of S_1's postcondition that the break part of $S_1; S_2$ holds. For the same reason, the return part of S_1's postcondition is special.

The Q:LOOP rule uses the same principles as the Q:SEQ rule to tweak the final postcondition. In the case of Q:LOOP, we simply ensure that the break part of the inner statement becomes the skip part of the overall statement. We use $\bot$ as the break part of the loop S statement since its operational semantics prevent it from terminating differently than with a skip or a return.

The Q:CALL rule accounts for the actual stack-space usage of programs. It enforces that enough stack space is available to call the function f by adding $M(f)$ to the pre- and postcondition. The pre- and postconditions are taken from the function context Γ.

There are two weakening rules in the quantitative Hoare logic. The framing rule Q:FRAME weakens a statement by stating that if S needs P bytes to run and leaves Q bytes free at its end, then it can very well run with $P + c$ bytes and return $Q + c$ bytes. It is very handy to prove tight bounds using the max function as demonstrated in Figure 5. The consequence Q:CONSEQ rule is directly imported from classical Hoare logics except that instead of using the logical implication $\Rightarrow$ we use the quantitative $\geqslant$.

Auxiliary State The main difference between the implemented logic and the logic described here is that the latter does not have an *auxiliary state*. Auxiliary state is a classic extension of Hoare logic (see for example [30]). The auxiliary state is used to share information between the pre- and postcondition of a triple. In a logic without auxiliary state (or similar techniques) it is not possible to relate program states before and after a statement. For example, you cannot specify that the function `int twice () { i = i+i;}` doubles the value of the variable i. With an auxiliary variable Z it is possible specify this fact in Hoare logic using the triple $\{i = Z\}\,\mathsf{twice}()\,\{i = 2 \cdot Z\}$.

One technical challenge with this auxiliary state is that some triples, for example $\{i = Z\}\,c\,\{i = Z\}$ and $\{i = Z - 1\}\,c\,\{i = Z - 1\}$ need to be proved equivalent by the logic to handle recursive calls. This problem is usually solved by introducing a more complex consequence rule, which our implemented system features. The typical use case is when we apply the rule Q:CALL to a recursive call. In this case, the Hoare triple for the function call is proved by an assumption from the derivation context with a slightly different auxiliary state. In the example derivation in Figure 6 this different state is $Z - 1$. Adapting the derivation hypothesis to prove the recursive call is enabled in our logic by an extended consequence rule that we proved sound in the quantitative setting.

Stack Framing Another minor difference is in the function application rule where we only present the rule for function calls with a single argument and without *framing of stack assertions*. The latter is necessary to carry over information on the local environment from the precondition to the postcondition of the function call.

$\{Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \Rightarrow M_b \cdot Z\}$
```
bsearch(x,l,h) {
  if (h-l <= 1) return l;
```
$\{(Z{>}0 \wedge Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma)) \Rightarrow M_b \cdot Z\}$
```
  m = (h+l)/2;
```
$\{(Z{>}0 \wedge Z = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \wedge \mathsf{m}_\sigma = \frac{\mathsf{h}_\sigma + \mathsf{l}_\sigma}{2}) \Rightarrow M_b \cdot Z\}$
```
  if (a[m]>x) h=m else l=m;
```
$\{[Z{-}1 = \log_2(\mathsf{h}_\sigma - \mathsf{l}_\sigma) \Rightarrow M_b \cdot (Z{-}1)] + M_b\}$
```
  return bsearch(x,l,h);
```
$\{[M_b \cdot (Z{-}1)] + M_b\}$
```
}
```
$\{M_b \cdot Z\}$

Figure 6. Derivation with auxiliary state for the bsearch function.

Soundness The soundness of our quantitative logic can be simply expressed by the following theorem.

Theorem 2. *For a fixed global environment* (Σ, Δ)*, a derivation in our quantitative logic for a statement* S *implies a bound on the weight of* S*, that is,*

$$\cdot \vdash \{P\}\,S\,\{Q\} \implies \forall \sigma, M\,.\,P(\sigma,M) \geqslant W_{(\sigma,M)}(S, \mathsf{Kstop}).$$

Naturally, we have to prove a stronger statement that takes postconditions and continuations into account to justify the soundness of the rules of the logic. This is not unlike as in program logics for low-level code [22] and Hoare-style logics for CompCert Clight [3]. Furthermore, we have to assume that we have a non-empty function context Γ; and finally, we have to step-index the correctness statement in order to prove its soundness by induction. The details can be found in the TR [9] and in the Coq development. Of course we prove in Coq that the intuitive validity, as formulated in Theorem 2 is a consequence of our stronger formulation of validity.

Example Figure 5 contains an example derivation for the statement $\mathsf{f}(); \mathsf{g}()$ in our logic. We assume that we have already verified that the function bodies of f and g do not allocate stack space, that is, $\Gamma(g) = \Gamma(f) = (\lambda(v,H)\,.\,0, \lambda(v,H)\,.\,0)$.

Our goal is to derive a quantitative Hoare triple that expresses that $\max(m_f, m_g)$, the maximum of the stack frame sizes of f and g, is a bound on the stack usage; and that after the execution $\max(m_f, m_g)$ stack space is available. Since the effect of break and return statements cannot leak outside of a function body, the corresponding postconditions can be arbitrary and we simply use $\bot$.

To derive our goal, we first have to apply the rule Q:SEQ for sequential composition. In the derivation of the function call $\mathsf{f}()$, we first reorder the precondition to get it in a form in which we can apply the rule Q:FRAME to eliminate the max operator. We then have a triple that is amenable to an application of the rule Q:CALL that uses the specification of the body of f in Γ.

4.4 Limitations

In our program logic described in this section, we do not consider function pointers, goto statements, continue statements, and switch statements, even though our Quantitative CompCert compiler still supports all of these. It would be possible to add these features to our logic by building on the ideas of advanced program logics like XCAP [29].

5. Automatic Stack Analyzer

In larger C programs a manual, interactive verification with a program logic is too tedious and time-consuming to be practical. Therefore we have developed an automatic stack analysis tool that operates at the Clight level to enable the analysis of real system code. We view this automatic tool mainly as a proof of concept that demonstrates the value of the logic for formal verification of static analysis tools. In the future, we will extend our automatic analyzer with advanced techniques like amortized resource analysis [5, 21]. This is however beyond the scope of this article.

The basic idea of our automatic stack analyzer is to compute a call graph from the Clight code and to derive a stack bound for each function in topological order. In Coq, the derivation of a function bound is implemented by a recursive function auto_bound on the abstract syntax tree (AST) of a Clight program. The function auto_bound does not only compute a stack bound but also a derivation in our quantitative program logic. This verifies the correctness of the generated bound and enables the composition of stack bounds that have been derived interactively or with other static analysis tools. In addition to the AST, auto_bound takes a context of known function bounds together with their derivations in the logic as an argument.

Given our verified quantitative logic, the implementation of auto_bound is straightforward. For trivial commands like assignments or skip, auto_bound simply generates the bound 0 and a derivation like $\{0\}\,\mathsf{skip}\,\{(0,0,0)\}$. For a sequential composition $S_1; S_2$ we inductively apply auto_bound to S_1 and S_2, and derive the bounds $\{B_i\}\,S_i\,\{(B_i^s, B_i^b, B_i^r)\}$ for $i{=}1,2$. We then return the precondition $\max\{B_1, B_2\}$ and the postcondition $(\max\{B_1^s, B_2^s\}, \max\{B_1^b, B_2^b\}, \max\{B_1^s, B_2^s\})$ for the command $S_1; S_2$. The derivation of this bound is similar to the example derivation that is sketched in Figure 5. The computation of the bound for the conditional works similar. For loops we can use the bound derived for the loop body to obtain a bound for the loop. In the derivation we just apply the rule Q:LOOP. Function calls are handled with the context of known function bounds (recursion is not allowed here) and the rule Q:CALL.

We envision, that the quantitative logic can be a useful backend to verify more sophisticated static analyses. For our simple, automatic stack analyzer the logic was already very convenient and enabled us to verify the analyzer almost without additional effort.

We have combined our automatic stack analyzer with our Quantitative CompCert compiler. The result is a verified C compiler that translates a program without function pointers and recursive calls to x86 assembly and automatically derives a stack bound for each function in the program including main(). The soundness theorem we have proved states the following. If a given program is memory-safe and the verified compiler successfully produces an assembly program A then A refines the source program and runs safely on an x86 machine with the stack size that has been computed by the automatic stack analysis for main() (see Point 3 of Theorem 1).

6. Experimental Evaluation

To validate the practicality of our framework for stack-bound verification, we have performed an experimental evaluation with more than 3000 lines of C code from different sources including hand written code, programs from the CompCert test suite, programs from the MiBench [17] embedded software benchmark suite, and modules from the simplified development version of the CertiKOS operating system kernel which is currently being verified.

File Name / Line Count	Function Name	Verified Stack Bound
mibench/net/dijkstra.c	enqueue	40 bytes
(174 LOC)	dequeue	40 bytes
	dijkstra	88 bytes
mibench/auto/bitcount.c	bitcount	16 bytes
(110 LOC)	bitstring	32 bytes
mibench/sec/blowfish.c	BF_encrypt	40 bytes
(233 LOC)	BF_options	8 bytes
	BF_ecb_encrypt	80 bytes
mibench/sec/pgp/md5.c	MD5Init	16 bytes
(335 LOC)	MD5Update	168 bytes
	MD5Final	168 bytes
	MD5Transform	128 bytes
mibench/tele/fft.c	IsPowerOfTwo	16 bytes
(195 LOC)	NumberOfBitsNeeded	24 bytes
	ReverseBits	24 bytes
	fft_float	160 bytes
certikos/vmm.c	palloc	48 bytes
(608 LOC)	pfree	40 bytes
	mem_init	72 bytes
	pmap_init	176 bytes
	pt_free	80 bytes
	pt_init	152 bytes
	pt_init_kern	136 bytes
	pt_insert	80 bytes
	pt_read	56 bytes
	pt_resv	120 bytes
certikos/proc.c	enqueue	48 bytes
(819 LOC)	dequeue	48 bytes
	kctxt_new	72 bytes
	sched_init	232 bytes
	tdqueue_init	208 bytes
	thread_init	192 bytes
	thread_spawn	96 bytes
compcert/mandelbrot.c	main	56 bytes
(92 LOC)		
compcert/nbody.c	advance	80 bytes
(174 LOC)	energy	56 bytes
	offset_momentum	24 bytes
	setup_bodies	16 bytes
	main	112 bytes

Table 1. Automatically verified stack bounds for C functions.

Tables 1 and 2 show a representative compilation of the experiments. Table 1 contains bounds that were automatically derived with the stack analyzer. Table 2 contains 8 bounds that were interactively derived using the quantitative logic with occasional support of the automation. The size of the analyzed example files varies from 8 lines of code (fib.c) to 819 lines of code (proc.c). In general, the automatic stack-bound analysis runs very efficiently and needs less than a second for every example file on (one core of) a Linux workstation with 32G of RAM and a x86 processor with 16 cores at 3.10Ghz.

In Table 1, the first column shows the file name of the examples together with the number of lines, the second column contains the name of selected functions from that file, and the third column contains the verified bound. The interactively-derived bounds in Table 2 are presented as symbolic expressions parametric in the functions'

Function Name	Verified Stack Bound
recid()	8a bytes
bsearch(x, lo, hi)	$40(1 + \log_2(\mathsf{hi} - \mathsf{lo}))$ bytes
fib(n)	24n bytes
qsort(a, lo, hi)	$48(\mathsf{hi} - \mathsf{lo})$ bytes
filter_pos(a, sz, lo, hi)	$48(\mathsf{hi} - \mathsf{lo})$ bytes
sum(a, lo, hi)	$32(\mathsf{hi} - \mathsf{lo})$ bytes
fact_sq(n)	$40 + 24\mathsf{n}^2$ bytes
filter_find(a, sz, lo, hi)	$128 + 48(\mathsf{hi} - \mathsf{lo}) + 40\log_2(\mathsf{BL})$ bytes

Table 2. Manually verified stack bounds for C functions.

arguments. These symbolic expressions are slight simplifications of the real pre- and postconditions of the functions that we proved in Coq. The actual Hoare triples proved in Coq carry a logical meaning which does, for instance, require that the qsort function be called on a valid sub array. The file sizes of the manual verified examples range from 8 to 52 lines of code.

Our main application of the automatic stack-analyzer is the CertiKOS operating system kernel [15]. Currently, the stack in CertiKOS is preallocated and proving the absence of stack-overflow is essential in the verification of the reliability of the system. Since CertiKOS does not use recursion, we can use the automatic analysis to derive precise stack bounds. Using our Quantitative CompCert compiler, we were, for instance, able to compile and compute bounds for the virtual memory management module (certikos/vmm.c) and the process management module (certikos/proc.c). Because of the large number of functions in CertiKOS, only a sample of the analyzed functions is displayed in Table 1.

Testing the quantitative Hoare logic and the compiler on CompCert test suite was a natural choice since our compiler builds on CompCert's architecture. This also allowed us to make sure that we did not introduce any regression with respect to the original CompCert compiler. To stress the expressivity of the logic we focused on test programs with recursive functions. The functions fib and qsort in Table 2 are for instance from the CompCert test suite. Files with automatically derived bounds for non-recursive functions from the CompCert test suite include mandelbrot.c which computes an approximation of the Mandelbrot set and nbody.c which computes an n-body simulation of a part of our solar system.

We also made sure that our technique can handle safety critical embedded software. The MiBench [17] benchmark that we used for this purpose is free, publicly available, and representative for embedded software. The use of recursion in MiBench programs is relatively rare, which makes them a great target for our automatic stack analyzer. The analyzed examples we present in Table 1 include for instance Dijkstra's single-source shortest-path algorithm (dijkstra.c), and the cryptographic algorithms Blowfish (blowfish.c) and MD5 (md5.c).

Finally, Table 2 contains some recursive functions that demonstrate the expressivity of our quantitative logic. The function bsearch is, for example, a recursive binary search with logarithmic recursion depth. The function fib computes the Fibonacci sequence using an exponential algorithm and the function qsort implements a recursive version of the quicksort algorithm. The verification of the function fact_sq shows the modularity of the logic: We first verify a linear bound for the factorial function and then use this bound to verify fact_sq(n), which contains the call fact(n^2). The function filter_pos takes an array and computes a new array that contains all positive elements of the input array. Similarly, filter_find uses the binary search bsearch to filter out all elements of an input array that are contained in another array of size BL. The modularity of the logic enables us to reuse the logarithmic bound that we already derived for bsearch in the proof. The verification of some functions is still

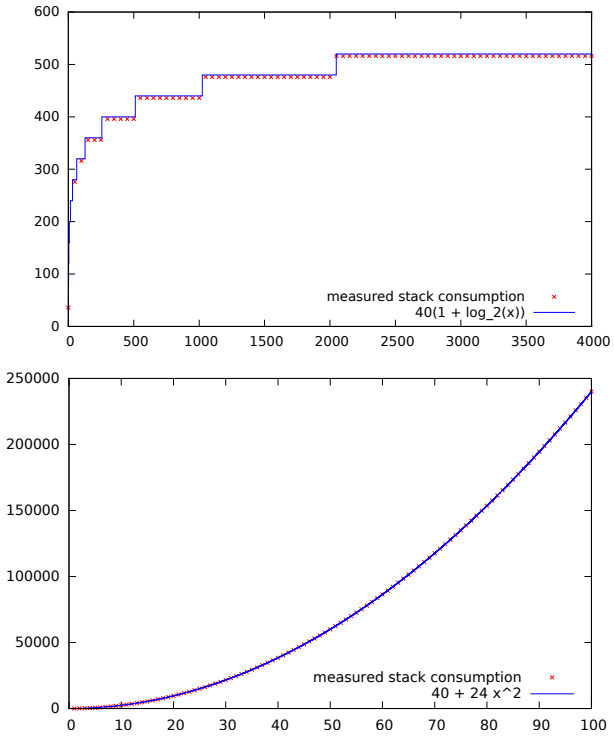


Figure 7. Experimental evaluation of the accuracy of hand-derived stack bounds. The plots compare the derived bounds (blue lines) for the functions bsearch (at the top) and fact_sq (at the bottom) with the measured stack usage of the execution of the respective function for different inputs (red crosses). The x-axis shows either the value of an integer argument (fact_sq) or the length of an input array (bsearch). The y-axis shows the stack usage in bytes.

underway. The bounds for the functions recid, bsearch, fib, and qsort are already completely verified.

Our experiments show that the automatic stack analyzer works effectively for our main application, the CertiKOS OS kernel. The reason is that we designed the quantitative logic to include exactly the subset of Clight that is needed for CertiKOS. It turned out that this subset is also sufficient for many examples in the CompCert test suite and the MiBench embedded software benchmarks. If a program is not interactively analyzable in our logic then this is due to unsupported language constructs such as switch statements and function pointers. Many of these language features could easily be supported by relatively small additions to the logic. An exception to this are function pointers which would require more work, following for example XCAP [29].

We have evaluated the accuracy of the verified bounds by comparing them with the actual stack-space consumption of the compiled C programs. Our experiments show that our framework is expressive enough to derive very tight bounds for recursive and non-recursive programs. All manually and automatically derived bounds over-approximate the actual stack-space consumption by exactly 4 bytes. Figure 7 shows the results of two representative experiments with hand-derived bounds for recursive programs. The bound derived in the logic is plotted together with the actual stack consumption of C programs measured on different inputs.

Measuring the stack consumption of C programs on modern computers is not as trivial as we originally thought. The measurement is complicated by some security mechanisms and unrestricted manipulation of the stack pointer by the compiler. To this end, we

designed a small C program that uses the linux system call `ptrace`. Using the this system call our tool forks the monitored process as a child then executes it step by step while keeping track of its stack consumption.

The reason for the 4-byte looseness of the bounds is that stack frames always reserve four bytes for a potential function call: The return address needs to be pushed by a call instruction in the callee. Obviously, the last function in the function call chain does not call any other function. So these four bytes remain unused.

7. Related Work

We now discuss research that is related to our contributions in verified compilation, program logics, and automatic resource analysis.

Verified Compilation Soundness proofs of compilers have been extensively studied and we focus on *formally verified* proofs here. Klein and Nipkow [24] developed a verified compiler from an object-oriented, Java-like language to JVM byte code. Chlipala [11] describes a verified compiler from the simply-typed lambda calculus to an idealized assembly language. In contrast to our work, the aforementioned works do not model nor preserve quantitative properties such as stack usage.

Our verified Quantitative CompCert compiler is an extension of the CompCert C Compiler [26, 27]. Despite being formally verified, important quantitative properties such as memory and time usage of programs compiled with CompCert have still to be verified at the assembly level [6]. Admittedly, there exists a clever annotation mechanism [6] in CompCert that allows to transport assertions on program states from the source level to the target machine code. However, these assertions can only contain statements about memory states but not bounds on the number of loop iterations and or recursion depth of functions. The novelty of our Quantitative CompCert extension to CompCert is that it enables us to reason about quantitative properties of event traces during compilation. Another novelty is that we model the assembly level semantics more realistically by using a finite stack. In particular, we do not have to use pseudo instructions anymore. This is similar to CompCertTSO [32]. However, we use event traces to get guarantees on the size of the stack that is needed to ensure refinement. On the other hand, it is always possible that the compiled code runs out of stack space in CompCertTSO.

In the context of the Hume language [18], Jost et al. [23] developed a quantitative semantics for a functional language and related it to memory and time consumption of the compiled code for the Renesas M32C/85U embedded micro-controller architecture. In contrast to our work, the relation of the compiled code with functional code is not formally proved.

Program Logics In the development of our quantitative Hoare logic we have drawn inspiration from mechanically verified Hoare logics. Nipkow's [30] description of his implementations of Hoare logics in Isabelle/HOL has been helpful to understand the interaction of auxiliary variables with the consequence rule. The consequence rule we use in our Coq implementation is a quantitative version of a consequence rule that has been attributed to Martin Hofmann by Nipkow [30]. Appel's separation logic for CompCert Clight [3] has been a blueprint for the structure of the quantitative logic. Since we do not deal with memory safety, our logic is much simpler and it would be possible to integrate it with Appel's logic. The continuation passing style that we use in the quantitative logic is not only used by Appel [3] but also in Hoare logics for low-level code [22, 29].

There exist quantitative logics that are integrated into separation logic [5, 20] and they are closely related to our quantitative logic. However, the purpose of these logics is slightly different since they focus on the verification of bounds that depend on the shape of heap data structures. Moreover, they are only defined for idealized languages and do not provide any guarantees for compiled code. Also closely related to our logic is a VDM-style logic for reasoning about resource usage of JVM byte code by Aspinall et al. [4]. Their logic is more general and applies to different quantitative resources while we focus on stack usage. However, it is unclear how realistic the presented resource metrics are. On the other hand, our logic applies to system code written in C, is verified with respect to CompCert Clight, and derives bounds for x86 assembly.

Resource Analysis There exists a large body of research on statically deriving stack bounds on low-level code [8, 10, 31] as well as commercial tools such as the *Bound-T Time and Stack Analyser* and Absint's *StackAnalyzer* [14]. We are however not aware of any formally verified techniques. For high-level languages there exists a large number of systems for statically inferring or checking quantitative requirements such as stack usage [1, 12, 19, 23]. However, they are not formally verified and do not apply to system code that is written in C. For C programs, there exist methods to automatically derive loop bounds [16, 36] but the proposed methods are not verified and it is unclear if they can be used for computing stack bounds.

We are only aware of two verified quantitative analysis systems. Albert et al. [2] rely on the KeY tool to automatically verify previously inferred loop invariants, size relations, and ranking functions for Java Card programs. However, they do not have a formal cost semantics and do not verify actual stack bounds. Blazy et al. [7] have verified a loop bound analysis for CompCert's RTL intermediate language. It is however unclear how the presented technique can be used to verify stack bounds or to formally translate bounds to a lower-level during compilation.

8. Conclusion

Embedded software has always been a target of verified compilers. As a result, aiding verification of quantitative properties remains a major goal for verified compilation. In one of the earliest articles [26] on CompCert, Leroy stated:

> "[...] it is hopeless to prove a stack memory bound on the source program and expect this resource certification to carry out to compiled code: stack consumption, like execution time, is a program property that is not preserved by compilation."

Ironically, Leroy's groundbreaking work on CompCert has been the main inspiration in our development of a framework that enables exactly such a resource certification of stack-consumption bounds for compiled x86 assembly code *at the C level*.

We have developed Quantitative CompCert, a realistic, verified C compiler which shows how verified compilation enables the verification of quantitative properties of compiled programs at the source level. We have implemented and formally verified a novel quantitative Hoare logic for CompCert Clight which is an ideal backend for static analysis tools. This is demonstrated through the implementation of a verified, automatic stack-analysis tool that computes derivations in the quantitative logic. Finally, we have shown through experiments that our framework can be applied to derive precise stack bounds for typical system code.

Our work opens the door for the verification of powerful static analysis tools for quantitative properties that operate on the C level rather than on the machine code. There are multiple future research directions that we plan to explore on the basis of the present development. For one thing, we want to use our quantitative Hoare logic to verify more powerful analysis tools that can automatically derive stack-space bounds for recursive functions. For another thing, we plan to generalize the developed concepts to apply our technique to other resources such as heap-memory and clock-cycle consumption.

Acknowledgments

We thank Lennart Beringer, Francesco Logozzo, the anonymous reviewers of PLDI'14, and the PLDI'14 Artifact Evaluation Committee for helpful comments and suggestions that improved this article and the implemented tools.

This research is based on work supported in part by NSF grants 1319671 and 1065451, and DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO Programs. In *Prog. Langs. and Systems - 9th Asian Symposium (APLAS'11)*, pages 238–254, 2011.

[2] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Fundamental Approaches to Soft. Eng. - 15th Int. Conf. (FASE'12)*, pages 130–145, 2012.

[3] A. W. Appel et al. *Program Logics for Certified Compilers*. Cambridge University Press, 2013.

[4] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A Program Logic for Resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[5] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[6] R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally Verified Optimizing Compilation in ACG-based Flight Control Software. In *Embedded Real Time Software and Systems (ERTS 2012)*, 2012.

[7] S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th Int. Conf. (VSTTE'13)*, 2013. To appear.

[8] D. Brylow, N. Damgaard, and J. Palsberg. Static Checking of Interrupt-Driven Software. In *23rd Int. Conf. on Soft. Engineering (ICSE'01)*, pages 47–56, 2001.

[9] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. Technical Report YALEU/DCS/TR-1487, Yale University, March 2014.

[10] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *7th Int Symp. on Memory Management (ISMM'08)*, pages 151–160, 2008.

[11] A. Chlipala. A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language. In *28th Conf. on Prog. Lang. Design and Impl. (PLDI'07)*, pages 54–65, 2007.

[12] K. Crary and S. Weirich. Resource Bound Certification. In *27th ACM Symp. on Principles of Prog. Langs. (POPL'00)*, pages 184–198, 2000.

[13] Express Logic, Inc. Helping you avoid stack overflow crashes! White Paper, 2014. URL `http://rtos.com/images/uploads/Stack_Analysis_White_paper.1_.pdf`.

[14] C. Ferdinand, R. Heckmann, and B. Franzen. Static Memory and Timing Analysis of Embedded Systems Code. In *3rd Europ. Symp. on Verification and Validation of Software Systems (VVSS'07)*, 2007.

[15] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Asia Pacific Workshop on Systems (APSys'11)*, 2011.

[16] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *IEEE International Workshop on Workload Characterization (WWC'01)*, pages 3–14, 2001.

[18] K. Hammond and G. Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Generative Progr. and Component Eng., 2nd Int. Conf. (GPCE'03)*, pages 37–56, 2003.

[19] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[20] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative Reasoning for Proving Lock-Freedom. In *28th ACM/IEEE Symposium on Logic in Computer Science (LICS'13)*, 2013.

[21] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[22] J. B. Jensen, N. Benton, and A. Kennedy. High-Level Separation Logic for Low-Level Code. In *40th ACM Symp. on Principles of Prog. Langs. (POPL'13)*, pages 301–314, 2013.

[23] S. Jost, H.-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. Carbon Credits for Resource-Bounded Computations using Amortised Analysis. In *16th Symp. on Form. Meth. (FM'09)*, pages 354–369, 2009.

[24] G. Klein and T. Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[25] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an Operating-System Kernel. *Commun. ACM*, 53(6):107–115, 2010.

[26] X. Leroy. Formal Certification of a Compiler Back-End, or: Programming a Compiler with a Proof Assistant. In *33rd Symposium on Principles of Prog. Langs. (POPL'06)*, pages 42–54, 2006.

[27] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[28] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50–57, 2013. ISSN 0740-7459.

[29] Z. Ni and Z. Shao. Certified Assembly Programming with Embedded Code Pointers. In *33th ACM Symp. on Principles of Prog. Langs. (POPL'06)*, pages 320–333, 2006.

[30] T. Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, volume 62 of *NATO Science Series*, pages 341–367. Springer, 2002.

[31] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, 2005.

[32] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60(3), 2013.

[33] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[34] R. Wilhelm et al. The Worst-Case Execution-Time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[35] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *32nd Conf. on Prog. Lang. Design and Impl. (PLDI'11)*, pages 283–294, 2011.

[36] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symposium (SAS'11)*, 2011.

Compositional Verification of Termination-Preserving Refinement of Concurrent Programs

Hongjin Liang[†] Xinyu Feng[†] Zhong Shao[‡]

[†]University of Science and Technology of China [‡]Yale University

lhj1018@mail.ustc.edu.cn xyfeng@ustc.edu.cn zhong.shao@yale.edu

Abstract

Many verification problems can be reduced to refinement verification. However, existing work on verifying refinement of concurrent programs either fails to prove the preservation of termination, allowing a diverging program to trivially refine any programs, or is difficult to apply in compositional thread-local reasoning. In this paper, we first propose a new simulation technique, which establishes termination-preserving refinement and is a congruence with respect to parallel composition. We then give a proof theory for the simulation, which is the first Hoare-style concurrent program logic supporting termination-preserving refinement proofs. We show two key applications of our logic, i.e., verifying linearizability and lock-freedom *together* for fine-grained concurrent objects, and verifying *full* correctness of optimizations of concurrent algorithms.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Correctness proofs, Formal methods; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Theory, Verification

Keywords Concurrency, Refinement, Termination Preservation, Rely-Guarantee Reasoning, Simulation

1. Introduction

Verifying refinement between programs is the crux of many verification problems. For instance, reasoning about compilation or program transformations requires proving that every target program is a refinement of its source [9]. In concurrent settings, recent work [4, 12] shows that the correctness of concurrent data structures and libraries can be characterized via some forms of contextual refinements, i.e., every client that calls the concrete library methods should refine the client with some abstract atomic operations. Verification of concurrent garbage collectors [11] and OS kernels [18] can also be reduced to refinement verification.

Refinement from the source program S to the target T, written as $T \sqsubseteq S$, requires that T have no more observable behaviors than S. Usually observable behaviors include the traces of external events such as I/O operations and runtime errors. The question is, should termination of the source be preserved too by the target? If yes, how to verify such refinement?

Preservation of termination is an indispensable requirement in many refinement applications. For instance, compilation and optimizations are not allowed to transform a terminating source program to a diverging (non-terminating) target. Also, implementations of concurrent data structures are often expected to have progress guarantees (e.g., lock-freedom and wait-freedom) in addition to linearizability. The requirements are equivalent to some contextual refinements that preserve the termination of client programs [12].

Most existing approaches for verifying concurrent program refinement, including simulations (e.g., [11]), logical relations (e.g., [22]), and refinement logics (e.g., [21]), do not reason about the preservation of termination. As a result, a program that does an infinite loop without generating any external events, e.g. `while true do skip`, would trivially refine any source program (just like that it trivially satisfies partial correctness in Hoare logic). Certainly this kind of refinement is not acceptable in the applications mentioned above.

CompCert [9] addresses the problem by introducing a well-founded order in the simulation, but it works only for sequential programs. It is difficult to apply this idea to do thread-local verification of concurrent program refinement, which enables us to know $T_1 \parallel T_2 \sqsubseteq S_1 \parallel S_2$ by proving $T_1 \sqsubseteq S_1$ and $T_2 \sqsubseteq S_2$. In practice, the termination preservation in the refinement proofs of individual threads could be easily broken by the interference from their environments (i.e., other threads running in parallel). For instance, a method call of a lock-free data structure (e.g., Treiber stack) may never terminate when other threads call the methods and update the shared memory infinitely often. As we will explain in Sec. 2, the key challenge here is to effectively specify the environments' effects on the termination preservation of individual threads. As far as we know, no previous work can use "compositional" thread-local reasoning to verify termination-preserving refinement between (whole) concurrent programs.

In this paper, we first propose novel rely/guarantee conditions which can effectively specify the interference over the termination properties between a thread and its environment. Traditional rely/guarantee conditions [8] are binary relations of program states and they specify the state updates. We extend them with a boolean tag indicating whether a state update may let the thread or its environment make more moves.

With the help of our new rely/guarantee conditions, we then propose a new simulation RGSim-T, and a new Hoare-style program logic, both of which support compositional verification of termination-preserving refinement of concurrent programs. Our work is based on our previous compositional simulation RGSim [11] (which unfortunately cannot preserve termination), and is inspired by Hoffmann et al.'s program logic for lock-

CSL-LICS 2014, July 14–18, 2014, Vienna, Austria.

http://dx.doi.org/10.1145/2603088.2603123

(a) source code S:

```
x++;
```

(b) target code T_b

```
1  local t;
2  t := x;
3  x := t + 1;
```

(c) target code T_c

```
1  local t, done := false;
2  while (! done) {
3    t := x;
4    done := cas(&x, t, t+1);
5  }
```

Figure 1. Counters.

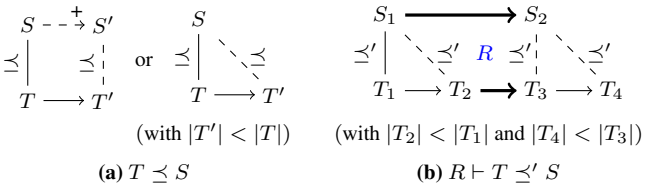


Figure 2. Simulation diagrams.

freedom [7] (which does not support refinement verification and has limitations on local reasoning, as we will explain in Sec. 7), but makes the following new contributions:

- We design a simulation, RGSim-T, to verify termination-preserving refinement of concurrent programs. As an extension of RGSim, it considers the interference between threads and the environments by taking our novel rely/guarantee conditions as parameters. RGSim-T is compositional. It allows us to thread-locally reason about the preservation of whole-program termination, but without enforcing the preservation of individual threads' termination, thus can be applied to many practical refinement applications.
- We propose the first program logic that supports compositional verification of termination-preserving refinement of concurrent programs. In addition to a set of compositionality (binary reasoning) rules, we also provide a set of unary rules (built upon the unary program logic LRG [3]) that can reason about conditional correspondence between the target and the source, a usual situation in concurrent refinement (see Sec. 2). The logic enables compositional verification of nested loops and supports programs with infinite nondeterminism. The soundness of the logic ensures RGSim-T between the target and the source, which implies the termination-preserving refinement.
- Our simulation and logic are general. They can be applied to verify linearizability and lock-freedom *together* for fine-grained concurrent objects, or to verify the full correctness of optimizations of concurrent programs, i.e., the optimized program preserves behaviors on both functionality and termination of the original one. We demonstrate the effectiveness of our logic by verifying linearizability and lock-freedom of Treiber stack [20], Michael-Scott queue [14] and DGLM queue [2], the full correctness of synchronous queue [16] and the equivalence between TTAS lock and TAS lock implementations [6].

It is important to note that our simulation and logic ensure that the target *preserves* the termination/divergence behaviors of the source. The target could diverge if the source diverges. Therefore our logic is not for verifying total correctness (i.e., partial correctness + termination). It is actually more powerful and general. We give more discussions on this point in Secs. 2.2 and 5.2.

In the rest of this paper, we first analyze the challenges and explain our approach informally in Sec. 2. Then we formulate termination-preserving refinement in Sec. 3. We present our new simulation RGSim-T in Sec. 4 and our new program logic in Sec. 5. We summarize examples that we verified in Sec. 6, and discuss the related work and conclude in Sec. 7.

2. Informal Development

Below we informally explain the challenges and our solutions in our design of the simulation and the logic respectively.

2.1 Simulation

Simulation is a standard technique for refinement verification. We start by showing a simple simulation for verifying *sequential* refinement and then discuss its problems in termination-preserving concurrent refinement verification.

Fig. 1(a) shows the source code S that increments `x`. In a sequential setting, it can be implemented as T_b in Fig. 1(b). To show that T_b refines S, a natural way is to prove that they satisfy the (weak) simulation $\preceq$ in Fig. 2(a).

The simulation first establishes some consistency relation between the source and the target (note S and T here are whole program configurations consisting of both code and states). Then it requires that there is some correspondence between the execution of the target and the source so that the relation is always preserved. Every execution step of the target must either correspond to one or more steps of the source (the left part of Fig. 2(a)), or correspond to zero steps (the right part; Let's ignore the requirement of $|T'| < |T|$ for now).[1]

For our example in Fig. 1, the simulation requires that `x` at the target level have the same value with `x` in the source. We let line `2` at T_b correspond to zero steps of S, and line `3` correspond to the single step of S.

Such a simulation, however, has two problems for termination-preserving concurrent refinement verification. First, it does not require the target to preserve the termination of the source. Since a silent step at the target level may correspond to zero steps at the source (the right part of Fig. 2(a)), the target may execute such steps infinitely many times and never correspond to a step at the source. For instance, if we insert `while true do skip` before line `2` in T_b, the simulation still holds, but T_b diverges now. To address this problem, CompCert [9] introduces a metric $|T|$ over the target program configurations, which is equipped with a well-founded order $<$. If a target step corresponds to no moves of the source, the metric over the target programs should strictly decrease (i.e., the condition $|T'| < |T|$ in Fig. 2(a)). Since the well-founded order ensures that there are no infinite decreasing chains, execution of the target will finally correspond to at least one step at the source.

Second, it is not compositional w.r.t. parallel compositions. Though $T_b \preceq S$ holds, $(T_b \parallel T_b); \mathbf{print}(x) \preceq (S \parallel S); \mathbf{print}(x)$ does not hold since the left side may print out 1, which is impossible for the source on the right. The problem is that when we prove $T_b \preceq S$, T_b and S are viewed as closed programs and the interference from environments is ignored. To get the parallel compositionality, we follow the ideas in our previous work RGSim [11] and parameterize the simulation with the interference between the programs and their environments.

As shown in Fig. 2(b), the new simulation $\preceq'$ is parameterized with the environment interference R, i.e. the set of all possible transitions of the environments at the target and source levels. Here we use thin arrows for the transitions of the current thread at the source and the target levels (e.g., from T_1 to T_2 and from T_3 to T_4 in Fig. 2(b)), and thick arrows for the possible environment steps (e.g., from T_2 to T_3 and from S_1 to S_2 in the figure). We require the simulation $\preceq'$ to be preserved by R.

[1] Note here we only discuss silent steps (a.k.a. τ-steps) which produce no external events. The simulation also requires that every step with an external event at the target level must correspond to one step at the source with the same event plus zero or multiple τ-steps.

Then, to prove termination-preserving concurrent refinement, it seems natural to combine the two ideas and have a simulation parameterized with environment interference and a metric decreasing for target steps that correspond to no steps at the source. Therefore we require $|T_2| < |T_1|$ and $|T_4| < |T_3|$ in the case of Fig. 2(b). But *how would the environment steps change the metric?*

First attempt. Our first attempt to answer this question is to allow environment steps to arbitrarily change the metrics associated with the target program configurations. Therefore it is possible to have $|T_2| < |T_3|$ in Fig. 2(b).

The resulting simulation, however, is still not compositional w.r.t. parallel compositions. For instance, for the following two threads in the target program:

```
while(i==0) i--;  ||  while(i==0) i++;
```

we can prove that this simulation holds between each of them and the source program `skip`, if we view `i` as local data used only at the target level. We could define the metric as 1 if `i` $= 0$ and 0 otherwise. For the left thread, it decreases the metric if it executes the loop body. The increment of `i` by its environment (the right thread) may change `i` back to 0, increasing the metric. This is allowed in our simulation. The case for the right thread is symmetric. However, if we view the parallel composition of the two threads as a whole program, it may not terminate, thus cannot be a termination-preserving refinement of `skip` $\|$ `skip`.

Second attempt. The first attempt is too permissive to have parallel compositionality, because we allow a thread to make more moves whenever its environment interferes with it. Thus our second attempt enforces the metric of a thread to decrease or stay unchanged under environment interference. For the case of Fig. 2(b), we require $|T_3| \leq |T_2|$ on environment steps.

This simulation is compositional, but it is too strong and cannot be satisfied by many useful refinements. For instance, T_c in Fig. 1(c) uses a compare-and-swap (`cas`) instruction to atomically update `x`. It is a correct lock-free implementation of S in concurrent settings, but the new simulation of our second attempt does not hold between T_c and S. If an environment step between lines 3 and 4 of T_c increments `x`, the `cas` at line 4 will return false and T_c needs to execute another round of loop. Therefore such an environment step increases the number of silent steps of T_c that correspond to no moves of S. However, our new simulation does not allow an environment step to increase the metric, so the simulation cannot be established.

Our solution. Our solution lies in the middle ground of the two failed attempts. We specify explicitly in the parameter R which environment steps may make the current thread move more (i.e., allow the thread's metric to increase in the simulation). Here we distinguish in R the steps that correspond to source level moves from those that do not. We allow the metric to be increased by the former (as in our first attempt), but not by the latter (which must decrease or preserve the metric, as in our second attempt).

This approach is based on the observation that the failure of `cas` in T_c of Fig. 1(c) must be caused by an environment step that successfully increments `x`, which corresponds to a step at the source level. Although the termination of the current thread T_c is delayed, the whole system consisting of both the current thread and the environment progresses by making a corresponding step at the source level. Therefore, the delay of the termination of the current thread should be acceptable, and we should allow such environment steps to increase the metric of the current thread.

In this paper, we follow the idea of rely/guarantee reasoning [8] and use the rely condition to specify environment steps. However, we extend the traditional rely conditions with an extra boolean tag indicating whether an environment step corresponds to a step at the source level. Our new simulation RGSim-T extends RGSim by incorporating the idea of metrics to achieve termination preservation. It is parameterized with the new rely (and guarantee) conditions so that we know how an environment step could affect the metric. The formal definition of RGSim-T is given in Sec. 4.

Relationships to lock-freedom, obstruction-freedom and wait-freedom. If the source program is just a single atomic operation (e.g. `x++`), our new simulation RGSim-T can be viewed as a proof technique for lock-freedom of the target, which ensures that there always *exists* some thread that will complete an operation at the source level in a finite number of steps. That is, the failure of a thread to finish its operation must be caused by the successful completion of source operations by its environment.

In fact, the simulations of our first and second attempts can be viewed as proof techniques for obstruction-freedom and wait-freedom respectively of concurrent objects. Obstruction-freedom ensures that every thread will complete its operation whenever it is executed in isolation (i.e., without interference from other threads). In the simulation of our first attempt, though a thread is allowed to not make progress under environment interference, it has to complete some source operations when its environments do not interfere. Wait-freedom ensures the completion of the operation of any thread. Correspondingly in the simulation of our second attempt, a thread has to make progress no matter what the environment does.

2.2 Program Logic

The compositionality of our new simulation RGSim-T allows us to decompose the refinement for large programs to refinements for small program units, therefore we could derive a set of syntactic Hoare-style rules for refinement verification, as we did for RGSim [11]. For instance, a sequential composition rule may be in the following form:

$$\frac{R \vdash \{P\}T_1 \preceq S_1\{P'\} \qquad R \vdash \{P'\}T_2 \preceq S_2\{Q\}}{R \vdash \{P\}T_1; T_2 \preceq S_1; S_2\{Q\}}$$

Here we use $R \vdash \{P\}T \preceq S\{Q\}$ to represent the corresponding syntactic judgment of RGSim-T. R denotes the environment interference. P, Q and P' are relational assertions that relate the program states at the target and the source levels. The rule says if we could establish refinements (in fact, RGSim-Ts) between T_1 and S_1, and between T_2 and S_2, we know $T_1; T_2$ refines $S_1; S_2$. We could give similar rules for parallel composition and other compositional commands.

However, in many cases the correspondence between program units at the target and the source levels cannot be determined statically. That is, just by looking at $T_1; T_2$ and $S_1; S_2$, we may not know statically that T_1 refines S_1 and T_2 refines S_2 and then apply the above sequential composition rule. To see the problem, we unfold the while-loop of T_c in Fig. 1 and get the following T_c':

```
1  local t, done;          4  while (!done) {
2  t := x;                 5    t := x;
3  done := cas(&x,t,t+1);  6    done := cas(&x,t,t+1);
                           7  }
```

Clearly T_c' refines S too. However, whether the `cas` instruction at line 3 fulfils the operation in S or not depends on whether the comparison succeeds in runtime. Thus we cannot apply the compositionality rules for RGSim-T to decompose the refinement about T_c'. We have to refer to the semantics of the simulation definition to prove the refinement, which would be rather ineffective for large scale programs. Similar issues also show up in our earlier work on RGSim [11], and in relational Hoare logic [1] and relational separation logic [25] if they are applied to concurrent settings.

To address this problem, we extend the assertion language to specify as auxiliary state the source code remaining to be refined.

In addition to the binary judgment $R \vdash \{P\}T \preceq S\{Q\}$, we introduce a unary judgment in the form of $R \vdash \{P \wedge \mathsf{arem}(S)\}T\{Q \wedge \mathsf{arem}(S')\}$ for refinements that cannot be decomposed. Here $\mathsf{arem}(S)$ means S is the remaining source to be refined by the target. Then $R, G \vdash \{P \wedge \mathsf{arem}(S)\}T\{Q \wedge \mathsf{arem}(\mathtt{skip})\}$ says that T refines S, since the postcondition shows at the end of the target T there are no remaining operations from S to be refined. We provide the following rule to derive the binary judgment from the unary one:

$$\frac{R \vdash \{P \wedge \mathsf{arem}(S)\}T\{Q \wedge \mathsf{arem}(\mathtt{skip})\}}{R \vdash \{P\}T \preceq S\{Q\}}$$

On the other hand, if the final remaining source is the same as the initial one, we know the execution steps of the target correspond to zero source steps. Then for the T_c' above, we can give pre- and post-conditions for line 3 as follows:

$\{\cdots \wedge \mathsf{arem}(S)\}$
`done := cas(&x, t, t+1)`
$\{\cdots \wedge (\mathtt{done} \wedge \mathsf{arem}(\mathtt{skip}) \vee \neg \mathtt{done} \wedge \mathsf{arem}(S))\}$

As the postcondition shows, whether the `cas` instruction refines S or not is now conditional upon the value of `done`. Thanks to the new assertions $\mathsf{arem}(S)$, we can reduce the relational and semantic refinement proofs to unary and syntactic Hoare-style reasoning.

The key to verifying the preservation of termination is the rule for while loops. One may first think of the total correctness rule for while loops in Hoare-style logics (e.g., [19]). However, preserving the termination does not necessarily mean that the code must terminate, and the total correctness rule would not be applicable in many cases. For example, the following T_c'' and S' never terminate:

T_c'' :	S' :

`local t;`
`while (true){`
`  t := x;`
`  cas(&x, t, t+1);`
`}`
	`while (true)`
	`  x++;`

but $T_c'' \preceq S'$ holds for our RGSim-T ($\preceq$) — Every iteration of T_c'' either corresponds to a step of S', or is interfered by environment steps corresponding to source moves.

Inspired by Hoffmann et al.'s logic for lock-freedom [7], we introduce a counter n (i.e. the number of tokens assigned to the current thread) as a while-specific metric, which means the thread can only run the loop for no more than n rounds before it or its environment fulfils one or more source-level moves. The counter is treated as an auxiliary state, and decreases at the beginning of every round of loop (i.e., we pay one token for each iteration). If we reach a step in the loop body that corresponds to source moves, we could reset the counter to increase the number of tokens. Tokens could also increase under environment interference if the environment step corresponds to source moves. Correspondingly our WHILE rule is in the following form (we give a simplified version to demonstrate the idea here. The actual rule is given in Sec. 5):

$$\frac{P \wedge B \Rightarrow P' * \mathsf{wf}(1) \qquad R \vdash \{P'\}C\{P\}}{R \vdash \{P\}\textbf{while}\ (B)\ C\{P \wedge \neg B\}}$$

We use $\mathsf{wf}(1)$ to represent one token, and "$*$" for normal separating conjunction in separation logic. To verify the loop body C, we use the precondition P', which has one less token than P, showing that one token has been consumed to start this new round of loop. During the execution of C, the number of token could be increased if C itself or its environment steps correspond to source moves. As usual, the loop invariant P needs to be re-established at the end of C.

$$\begin{array}{llll}
(\textit{Event}) & e & ::= \ldots & (\textit{Label})\ \iota ::= e \mid \tau \\
(\textit{Store}) & s, \mathbb{s} & \in \textit{PVar} \rightharpoonup \textit{Val} & (\textit{Heap})\ h, \mathbb{h} \in \textit{Addr} \rightharpoonup \textit{Val} \\
(\textit{State}) & \sigma, \Sigma & ::= (s, h) & (\textit{Instr})\ c, \mathbb{c} ::= \ldots \\
(\textit{Expr}) & E, \mathbb{E} & ::= x \mid n \mid E + E \mid \ldots & \\
(\textit{BExp}) & B, \mathbb{B} & ::= \textbf{true} \mid \textbf{false} \mid E = E \mid !B \mid \ldots & \\
(\textit{Stmt}) & C, \mathbb{C} & ::= \textbf{skip} \mid c \mid \langle C \rangle \mid C_1; C_2 \mid \textbf{if}\ (B)\ C_1\ \textbf{else}\ C_2 & \\
 & & \mid \textbf{while}\ (B)\ C \mid C_1 \,\|\, C_2 &
\end{array}$$

Figure 3. Generic language at target and source levels.

To prove that T_c'' shown above preserves the termination of S', we set the initial number of tokens to 1. We use up the token at the first iteration, but could gain another token during the iteration (either by self moves or by environment steps) to pay for the next iteration. We can see that the above reasoning with tokens coincides with the direct refinement proof in our simulation RGSim-T. In fact, RGSim-T can serve as the meta-theory of our logic.

The use of tokens as an explicit metric for termination reasoning poses another challenge, which is to handle *infinite nondeterminism*. Consider the following target C.

C: `x := 0; while(i > 0) i--;`

Assume the environment R may arbitrarily update `i` when `x` is not 0, but does not change anything when `x` is 0. We hope to verify C refines `skip`. We can see that the loop in C must terminate (thus the refinement holds), and the number n of tokens must be no less than the value of `i` at the beginning of the loop. But we cannot decide the value of n before executing `x := 0`. This example cannot be verified if we have to predetermine and specify the metric for the while loops at the very beginning of the whole program.

To address this issue, we introduce the following hiding rule:

$$\frac{R \vdash \{p\}C\{q\}}{R \vdash \{\lfloor p \rfloor_{\mathtt{w}}\}C\{\lfloor q \rfloor_{\mathtt{w}}\}}$$

Here $\lfloor p \rfloor_{\mathtt{w}}$ discards all the knowledge about tokens in p. For the above example, we can hide the number of tokens after we verify the while loop. Then we do *not* need to specify the number of tokens in the precondition of the whole program. We formally present the set of logic rules in Sec. 5.

3. Formal Settings and Termination-Preserving Refinement

In this section, we define the termination-preserving refinement $\sqsubseteq$, which is the proof goal of our RGSim-T and logic.

3.1 The Language

Fig. 3 shows the programming language for both the source and the target levels. We model the program semantics as a labeled transition system. A label ι that will be associated with a state transition is either an event e or τ. The latter marks a silent step generating no events.

A state σ is a pair of a store and a heap. The store s is a finite partial mapping from program variables to values (e.g., integers and memory addresses) and a heap h maps memory addresses to values. Statements C are either primitive instructions or compositions of them. A single-step execution of statements is modeled as a labeled transition: $(C, \sigma) \xrightarrow{\iota} (C', \sigma')$. We abstract away the form of an instruction c. It may generate an external event (e.g., **print**(E) generates an output event). It may be non-deterministic (e.g., $x :=$ **nondet** assigns a random value to x). It may also be blocked at some states (e.g., requesting a lock). We assume primitive instructions are atomic in the semantics. We also provide an

$$\frac{(C,\sigma) \longrightarrow^{+} \textbf{abort}}{ETr(C,\sigma,\lightning)} \qquad \frac{(C,\sigma) \xrightarrow{e}{}^{+} (C',\sigma') \qquad ETr(C',\sigma',\mathcal{E})}{ETr(C,\sigma,e::\mathcal{E})}$$

$$\frac{(C,\sigma) \longrightarrow^{*} (\textbf{skip},\sigma')}{ETr(C,\sigma,\Downarrow)} \qquad \frac{(C,\sigma) \longrightarrow^{+} (C',\sigma') \qquad ETr(C',\sigma',\mathcal{E})}{ETr(C,\sigma,\mathcal{E})}$$

Figure 4. Co-inductive definition of $ETr(C,\sigma,\mathcal{E})$.

atomic block $\langle C\rangle$ to execute a piece of code C atomically. Then the generic language in Fig. 3 is expressive enough for the source and the target programs which may have different granularities of state accesses. Due to the space limit, the operational semantics and more details about the language are formally presented in TR [13].

Conventions. We usually write blackboard bold or capital letters ($\mathbb{s}$, $\mathbb{h}$, Σ, $\mathbb{c}$, $\mathbb{E}$, $\mathbb{B}$ and $\mathbb{C}$) for the notations at the source level to distinguish from the target-level ones (s, h, σ, c, E, B and C).

Below we use $\_ \longrightarrow^{*} \_$ for zero or multiple-step transitions with no events generated, $\_ \longrightarrow^{+} \_$ for multiple-step transitions without events, and $\_ \xrightarrow{e}{}^{+} \_$ for multiple-step transitions with *only one* event e generated.

3.2 Termination-Preserving Event Trace Refinement

Now we formally define the refinement relation $\sqsubseteq$ that relates the observable event traces generated by the source and the target programs. A trace $\mathcal{E}$ is a finite or infinite sequence of external events e, and may end with a termination marker $\Downarrow$ or an abortion marker $\lightning$. It is co-inductively defined as follows.

$$(\textit{EvtTrace}) \quad \mathcal{E} \quad ::= \quad \Downarrow \mid \lightning \mid \epsilon \mid e::\mathcal{E} \quad \text{(co-inductive)}$$

We use $ETr(C,\sigma,\mathcal{E})$ to say that the trace $\mathcal{E}$ is produced by executing C from the state σ. It is co-inductively defined in Fig. 4. Here **skip** plays the role of a flag showing the end of execution (the normal termination). Unsafe executions lead to **abort**. We know if C diverges at σ, then its trace $\mathcal{E}$ is either of infinite length or finite but does not end with $\Downarrow$ or $\lightning$. For instance, **while** (**true**) **skip** only produces an empty trace ϵ, and **while** (**true**) $\{\textbf{print}(1)\}$ only produces an infinite trace of output events.

Then we define a refinement $(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$, saying that every event trace generated by (C,σ) at the target level can be reproduced by $(\mathbb{C},\Sigma)$ at the source. Since we could distinguish traces of diverging executions from those of terminating executions, the refinement definition ensures that if (C,σ) diverges, so does $(\mathbb{C},\Sigma)$. Thus we know the target preserves the termination of the source.

Definition 1 (Termination-Preserving Refinement).
$(C,\sigma) \sqsubseteq (\mathbb{C},\Sigma)$ iff $\forall\mathcal{E}.\, ETr(C,\sigma,\mathcal{E}) \Longrightarrow ETr(\mathbb{C},\Sigma,\mathcal{E})$.

4. RGSim-T: A Compositional Simulation with Termination Preservation

Below we propose RGSim-T, a new simulation as a compositional proof technique for the above termination-preserving refinement. As we explained in Sec. 2, the key to compositionality is to parameterize the simulation with the interferences between the programs and their environments. In this paper, we specify the interferences using rely/guarantee conditions [8], but extend them to also specify the effects on the termination preservation of individual threads.

Our simulation relation between C and $\mathbb{C}$ is in the form of $R,G,I \models \{P\}C \preceq \mathbb{C}\{Q\}$. It takes R, G, I, P and Q as parameters. R and G are rely and guarantee conditions specifying the interference between the current thread and its environment. The assertion I specifies the consistency relation between states at the target and the source levels, which needs to be preserved during the execution. P specifies the pair of initial states at the target and the source levels from which the simulation holds, and Q is about the pair of final states when the target and the source terminate. So before we give our definition of RGSim-T, we first introduce our assertion language.

$$\begin{array}{rll}
(\textit{RelAssn}) & P,Q,I ::= & B \mid \mathsf{own}(x) \mid \mathsf{emp} \mid E \mapsto E \mid E \Mapsto E \\
& & \mid P * Q \mid P \vee Q \mid [\![p]\!] \mid \ldots \\
(\textit{FullAssn}) & p,q \;::= & P \mid \mathsf{arem}(\mathbb{C}) \mid \mathsf{wf}(E) \mid \lfloor p \rfloor_{\mathsf{a}} \mid \lfloor p \rfloor_{\mathsf{w}} \\
& & \mid p * q \mid p \vee q \mid \ldots \\
(\textit{RelAct}) & R,G \;::= & [P] \mid P \ltimes Q \mid P \propto Q \mid R{*}R \mid R^{+} \mid \ldots
\end{array}$$

Figure 5. Assertion language.

4.1 Assertions and New Rely/Guarantee Conditions

We show the syntax of the basic assertion language in Fig. 5, including the state assertions P and Q, and our new rely/guarantee conditions R and G (let's first ignore the assertions p and q, which will be explained in Sec. 5).

The state assertions P and Q relate the program states σ and Σ at the target and source levels. They are separation logic assertions over a pair of states. We show their semantics in the top part of Fig. 6. For simplicity, we assume the program variables used in the target code are different from the ones in the source (e.g., we use x and X for target and source level variables respectively). B holds if it evaluates to true at the disjoint union of the target and the source stores s and $\mathbb{s}$. We treat program variables as resources [15] and use $\mathsf{own}(x)$ for the ownership of the program variable x. The assertion $E_1 \mapsto E_2$ specifies a singleton heap of the *target* level with E_2 stored at the address E_1 and requires that the stores contain variables used to evaluate E_1 and E_2. Its counterpart for source level heaps is represented as $E_1 \Mapsto E_2$, whose semantics is defined similarly. emp describes empty stores and heaps at both levels. Semantics of separating conjunction $P * Q$ is similar as in separation logic, except that it is now lifted to assertions over relational states (σ,Σ). The union of two disjoint relational states (σ_1,Σ_1) and (σ_2,Σ_2) is defined in the middle part of Fig. 6. We will define the assertion $[\![p]\!]$ in Sec. 5 (see Fig. 8), which ignores the additional information other than the relational states about p.

Our new rely/guarantee assertions R and G specify the transitions over the relational states (σ,Σ) and also the effects on termination preservation. Their semantics is defined in the bottom part of Fig. 6. Here we use $\mathcal{S}$ for the relational states. A model consists of the initial relational state $\mathcal{S}$, the resulting state $\mathcal{S}'$, and an effect bit b to record whether the target transitions correspond to some source steps and can affect the termination preservation of the current thread (for R) or other threads (for G).

We use $[P]$ for identity transitions with the relational states satisfying P. The action $P \ltimes Q$ says that the initial relational states satisfy P and the resulting states satisfy Q. For these two kinds of actions, we do not care whether there is any source step in the transition satisfying them (the effect bit b in their interpretations could either be **true** or **false**). We also introduce a new action $P \propto Q$ asserting that one or more steps are made at the source level (the effect bit b must be **true**). Following LRG [3], we introduce separating conjunction over actions to locally reason about shared state updates. $R_1 * R_2$ means that the actions R_1 and R_2 start from disjoint relational states and the resulting states are also disjoint. But here we also require consistency over the effect bits for the two disjoint state transitions. We use R^{+} for the transitive closure of R, where the effect bits in consecutive transitions are accumulated. The syntactic sugars Id, Emp and True represent arbitrary identity transitions, empty transitions and arbitrary transitions respectively.

Since we logically split states into local and shared parts as in LRG [3], we need a precise invariant I to fence actions over shared

$((s,h),(\mathbb{s},\mathbb{h})) \models B$ iff $[\![B]\!]_{s \uplus \mathbb{s}} = \textbf{true}$

$((s,h),(\mathbb{s},\mathbb{h})) \models \mathsf{own}(x)$ iff $dom(s \uplus \mathbb{s}) = \{x\}$

$((s,h),(\mathbb{s},\mathbb{h})) \models E_1 \mapsto E_2$ iff $h = \{[\![E_1]\!]_{s \uplus \mathbb{s}} \rightsquigarrow [\![E_2]\!]_{s \uplus \mathbb{s}}\}$

$((s,h),(\mathbb{s},\mathbb{h})) \models \mathsf{emp}$ iff $s = h = \mathbb{s} = \mathbb{h} = \emptyset$

$f_1 \bot f_2$ iff $(dom(f_1) \cap dom(f_2) = \emptyset)$ $\quad f_1 \uplus f_2 \stackrel{\text{def}}{=} f_1 \cup f_2$, if $f_1 \bot f_2$

$(s_1,h_1) \bot (s_2,h_2)$ iff $(s_1 \bot s_2) \wedge (h_1 \bot h_2)$

$(s_1,h_1) \uplus (s_2,h_2) \stackrel{\text{def}}{=} (s_1 \cup s_2, h_1 \cup h_2)$, if $(s_1,h_1) \bot (s_2,h_2)$

$(\sigma_1,\Sigma_1) \uplus (\sigma_2,\Sigma_2) \stackrel{\text{def}}{=} (\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2)$, if $\sigma_1 \bot \sigma_2$ and $\Sigma_1 \bot \Sigma_2$

$\mathcal{S} ::= (\sigma, \Sigma)$

$(\mathcal{S},\mathcal{S}',b) \models [P]$ iff $(\mathcal{S} \models P) \wedge (\mathcal{S} = \mathcal{S}')$

$(\mathcal{S},\mathcal{S}',b) \models P \ltimes Q$ iff $(\mathcal{S} \models P) \wedge (\mathcal{S}' \models Q)$

$(\mathcal{S},\mathcal{S}',b) \models P \propto Q$ iff $(\mathcal{S} \models P) \wedge (\mathcal{S}' \models Q) \wedge (b = \textbf{true})$

$(\mathcal{S},\mathcal{S}',b) \models R_1 * R_2$ iff $\exists \mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_1',\mathcal{S}_2'.\ (\mathcal{S} = \mathcal{S}_1 \uplus \mathcal{S}_2) \wedge (\mathcal{S}' = \mathcal{S}_1' \uplus \mathcal{S}_2') \wedge ((\mathcal{S}_1,\mathcal{S}_1',b) \models R_1) \wedge ((\mathcal{S}_2,\mathcal{S}_2',b) \models R_2)$

$(\mathcal{S},\mathcal{S}',b) \models R^+$ iff $((\mathcal{S},\mathcal{S}',b) \models R) \vee (\exists \mathcal{S}'',b',b''.\ ((\mathcal{S},\mathcal{S}'',b') \models R) \wedge ((\mathcal{S}'',\mathcal{S}',b'') \models R^+) \wedge (b = b' \vee b''))$

$\mathsf{Id} \stackrel{\text{def}}{=} [\textbf{true}]$ $\quad \mathsf{Emp} \stackrel{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp}$ $\quad \mathsf{True} \stackrel{\text{def}}{=} \textbf{true} \ltimes \textbf{true}$

$I \triangleright R$ iff $([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I)$

$\mathsf{Sta}(P,R)$ iff $\forall \mathcal{S},\mathcal{S}',b.\ (\mathcal{S} \models P) \wedge ((\mathcal{S},\mathcal{S}',b) \models R) \Longrightarrow (\mathcal{S}' \models P)$

Figure 6. Semantics of assertions (part I).

states, which is a state assertion like P and Q. We define the fence $I \triangleright R$ in a similar way as in our previous work [10] and LRG [3], which says that I precisely determines the boundaries of the states of the transitions in R (see Fig. 6). The formal definition of the precise requirement $\mathsf{Precise}(I)$ is given in TR [13], which follows its usual meaning as in separation logic but is now interpreted over relational states.

4.2 Definition of RGSim-T

Our simulation RGSim-T is parameterized over the rely/guarantee conditions R and G to specify the interferences between threads and their environments, and a precise invariant I to logically determine the boundaries of the shared states and fence R and G.

The simulation also takes a metric M, which was referred to as $|T|$ in our previous informal explanations in Sec. 2. We leave its type unspecified here, which can be instantiated by program verifiers, as long as it is equipped with a well-founded order $<$.

The formal definition below follows the intuition explained in Sec. 2. Readers who are interested only in the proof theory could skip this definition, which can be viewed as the meta-theory of our program logic presented in Sec. 4.3 and Sec. 5.

Definition 2 (RGSim-T). $R,G,I \models \{P\}C \preceq \mathbb{C}\{Q\}$ iff
for all σ and Σ, if $(\sigma,\Sigma) \models P$, then there exists M such that $R,G,I \models (C,\sigma,M) \preceq_Q (\mathbb{C},\Sigma)$.

Here $R,G,I \models (C,\sigma,M) \preceq_Q (\mathbb{C},\Sigma)$ is the largest relation such that whenever $R,G,I \models (C,\sigma,M) \preceq_Q (\mathbb{C},\Sigma)$, then $(\sigma,\Sigma) \models I * \textbf{true}$ and the following are true:

1. for any C', σ'', σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \bot \Sigma_F$, then there exist σ', n, M', b, $\mathbb{C}'$ and Σ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$,
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^n (\mathbb{C}', \Sigma' \uplus \Sigma_F)$,
   (c) $R,G,I \models (C',\sigma',M') \preceq_Q (\mathbb{C}',\Sigma')$,
   (d) $((\sigma,\Sigma),(\sigma',\Sigma'),b) \models G^+ * \mathsf{True}$, and
   (e) if $n = 0$, we need $M' < M$ and $b = \textbf{false}$, otherwise $b = \textbf{true}$.
2. for any e, C', σ'', σ_F and σ_F, if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$ and $\Sigma \bot \Sigma_F$, then there exist σ', M', $\mathbb{C}'$ and Σ' such that
   (a) $\sigma'' = \sigma' \uplus \sigma_F$,
   (b) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \xrightarrow{e}{}^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$,
   (c) $R,G,I \models (C',\sigma',M') \preceq_Q (\mathbb{C}',\Sigma')$, and
   (d) $((\sigma,\Sigma),(\sigma',\Sigma'),\textbf{true}) \models G^+ * \mathsf{True}$.
3. for any b, σ' and Σ', if $((\sigma,\Sigma),(\sigma',\Sigma'),b) \models R^+ * \mathsf{Id}$, then there exists M' such that
   (a) $R,G,I \models (C,\sigma',M') \preceq_Q (\mathbb{C},\Sigma')$, and
   (b) if $b = \textbf{false}$, we need $M' = M$.
4. if $C = \textbf{skip}$, then for any Σ_F such that $\Sigma \bot \Sigma_F$, there exist n and Σ' such that
   (a) $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^n (\textbf{skip}, \Sigma' \uplus \Sigma_F)$,
   (b) $(\sigma, \Sigma') \models Q$,
   (c) if $n > 0$, then $((\sigma,\Sigma),(\sigma,\Sigma'),\textbf{true}) \models G^+ * \mathsf{True}$.
5. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \textbf{abort}$ and $\Sigma \bot \Sigma_F$, then $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \textbf{abort}$.

The simulation $R,G,I \models (C,\sigma,M) \preceq_Q (\mathbb{C},\Sigma)$ relates the executions of the target configuration (C,σ) (with its metric M) to the source $(\mathbb{C},\Sigma)$, under the interferences with the environment specified by R and G. It first requires that the relational state (σ,Σ) satisfy $I * \textbf{true}$, I for the shared part and **true** for the local part, establishing a consistency relation between the states at the two levels. For every silent step of (C,σ) (condition 1, let's first ignore the frame states σ_F and Σ_F which will be discussed later), the source could make n steps ($n \geq 0$) correspondingly (1(b)), and the simulation is preserved afterwards with a new metric M' (1(c)). Here we use $\_ \longrightarrow^n \_$ to represent n-step silent transitions. If $n = 0$ in 1(b) (i.e., the source does not move), the metric must decrease along the associated well-founded order ($M' < M$ in 1(e)), otherwise we do not have any restrictions over M'. We also require that the related steps at the two levels satisfy the guarantee condition $G^+ * \mathsf{True}$ (1(d)), the transitive closure G^+ for the shared part and True for the private. If the target step corresponds to *no* source moves ($n = 0$), we use **false** as the corresponding effect bit, otherwise the bit should be **true** (1(e)).

If a target step produces an event e, the requirements (condition 2) are similar to those in condition 1, except that we know for sure that target step corresponds to *one or more* source steps that produce the same e.

The simulation should be preserved after environment transitions satisfying $R^+ * \mathsf{Id}$, R^+ for the shared part and Id for the private (condition 3). If the corresponding effect bit of the environment transition is **true**, we know there are one or more source moves, therefore there are no restrictions over the metric M' for the resulting code (which could be larger than M). Otherwise, the metric should be unaffected under the environment interference (i.e., $M' = M$ in 3(b)).

If C terminates (condition 4), the corresponding $\mathbb{C}$ must also terminate and the resulting states satisfy the postcondition Q. Finally, if C is unsafe, then $\mathbb{C}$ must be unsafe too (condition 5).

Inspired by Vafeiadis [24], we directly embed the framing aspect of separation logic in Def. 2. At each condition, we introduce the frame states σ_F and Σ_F at the target and source levels to represent the remaining parts of the states owned by other threads in the system. The commands C and $\mathbb{C}$ must not change the frame states during their executions (see, e.g., conditions 1(a) and 1(b)). These σ_F and Σ_F quantifications in RGSim-T are crucial to admit the parallel compositionality and the frame rules (the B-FRAME rule in Fig. 7 and the FRAME rule in Fig. 9).

We then define $R,G,I \models \{P\}C \preceq \mathbb{C}\{Q\}$ by hiding the initial states via the precondition P and hiding the metric M.

$$\frac{R \vee G_2, G_1, I \vdash \{P_1 * P\} C_1 \preceq \mathbb{C}_1 \{Q_1 * Q_1'\} \quad R \vee G_1, G_2, I \vdash \{P_2 * P\} C_2 \preceq \mathbb{C}_2 \{Q_2 * Q_2'\} \quad P \vee Q_1' \vee Q_2' \Rightarrow I \quad I \triangleright R}{R, G_1 \vee G_2, I \vdash \{P_1 * P_2 * P\} C_1 \parallel C_2 \preceq \mathbb{C}_1 \parallel \mathbb{C}_2 \{Q_1 * Q_2 * (Q_1' \wedge Q_2')\}} \text{ (B-PAR)}$$

$$\frac{P \Rightarrow (B \Leftrightarrow \mathbb{B}) * I \quad R, G, I \vdash \{P \wedge B\} C \preceq \mathbb{C}\{P\}}{R, G, I \vdash \{P\} \textbf{while}\ (B)\ C \preceq \textbf{while}\ (\mathbb{B})\ \mathbb{C}\{P \wedge \neg B\}} \text{ (B-WHILE)} \qquad \frac{P \Rightarrow (E = \mathbb{E}) * I \quad \mathsf{Sta}(P, R * \mathsf{Id}) \quad I \triangleright \{R, G\}}{R, G, I \vdash \{P\} \textbf{print}(E) \preceq \textbf{print}(\mathbb{E}) \{P\}} \text{ (B-PRT)}$$

$$\frac{R, G, I \vdash \{P\} C \preceq \mathbb{C}\{Q\} \qquad \mathsf{Sta}(P', R' * \mathsf{Id}) \quad I' \triangleright \{R', G'\} \qquad P' \Rightarrow I' * \textbf{true} \qquad G^+ \Rightarrow G}{R * R', G * G', I * I' \vdash \{P * P'\} C \preceq \mathbb{C}\{Q * P'\}} \text{ (B-FRAME)}$$

Figure 7. Selected binary inference rules (compositionality of RGSim-T).

Adequacy. RGSim-T ensures the termination-preserving refinement by using the metric with a well-founded order. The proof of the following adequacy theorem is in TR [13].

Theorem 3 (Adequacy of RGSim-T). If there exist R, G, I, Q and a metric M (with a well-founded order $<$) such that $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.

4.3 Compositionality Rules

RGSim-T is compositional. We show some of the compositionality rules in Fig. 7. Here we use $R, G, I \vdash \{P\} C \preceq \mathbb{C}\{Q\}$ for the judgment to emphasize syntactic reasoning, whose semantics is RGSim-T (Def. 2). The rules can be viewed as the binary version of those in a traditional rely-guarantee-style logic (e.g., LRG [3] and RGSep [23]).

The B-PAR rule shows the compositionality w.r.t. parallel compositions. To verify $C_1 \parallel C_2$ is a refinement of $\mathbb{C}_1 \parallel \mathbb{C}_2$, we verify the refinement of each thread separately. The rely condition of each thread captures the interference from both the overall environment (R) and its sibling thread (G_1 or G_2). The related steps of $C_1 \parallel C_2$ and $\mathbb{C}_1 \parallel \mathbb{C}_2$ should satisfy either thread's guarantee. As in LRG [3], P_1 and P_2 specify the private (relational) states of the threads C_1/$\mathbb{C}_1$ and C_2/$\mathbb{C}_2$ respectively. The states P are shared by them. When both threads have terminated, their private states satisfy Q_1 and Q_2, and the shared states satisfy both Q_1' and Q_2'. We require that the shared states are well-formed (P, Q_1' and Q_2' imply I) and the overall environment transitions are fenced ($I \triangleright R$).

The B-WHILE rule requires the boolean conditions of both sides to be evaluated to the same value. The resources needed to evaluate them should be available in the private part of P. The B-FRAME rule supports local reasoning. The frame P' may contain shared and private parts, so it should be stable w.r.t. $R' * \mathsf{Id}$ and imply $I' * \textbf{true}$, where I' is the fence for R' and G' (see Fig. 6 for the definitions of fences and stability). We also require G to be closed over transitivity. This rule is almost identical to the one in LRG [3]. Details are elided here.

We provide a few binary rules to reason about the basic program units when they are almost identical at both sides. For instance, the B-PRT rule relates a target print command to a source one, requiring that they always print out the same value. For more general refinement units, as we explained in Sec. 2, we reduce relational verification to unary reasoning (using the U2B rule in Fig. 9, which we will explain in the next section). Our TR [13] contains more rules and the full soundness proofs. The soundness theorem is shown below.

Theorem 4 (Soundness of Binary Rules).
If $R, G, I \vdash \{P\} C \preceq \mathbb{C}\{Q\}$, then $R, G, I \models \{P\} C \preceq \mathbb{C}\{Q\}$.

5. A Rely-Guarantee-Style Logic for Termination-Preserving Refinement

The binary inference rules in Fig. 7 allow us to decompose the refinement verification of large programs into the refinement units' verification. In this section, we explain the unary rules for verifying refinement units. All the binary and unary rules constitute our novel rely-guarantee-style logic for modular verification of termination-preserving refinement.

$w \in \mathit{Nat} \qquad \mathbb{D} ::= \mathbb{C} \mid \bullet$

$(\sigma, w, \mathbb{D}, \Sigma) \models P$ iff $(\sigma, \Sigma) \models P$

$(\sigma, w, \mathbb{D}, \Sigma) \models \mathsf{arem}(\mathbb{C}')$ iff $\mathbb{D} = \mathbb{C}'$

$((s, h), w, \mathbb{D}, \Sigma) \models \mathsf{wf}(E)$ iff $\exists n. (\llbracket E \rrbracket_s = n) \wedge (n \leq w)$

$(\sigma, w, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{a}}$ iff $\exists \mathbb{D}'. (\sigma, w, \mathbb{D}', \Sigma) \models p$

$(\sigma, w, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{w}}$ iff $\exists w'. (\sigma, w', \mathbb{D}, \Sigma) \models p$

$(\sigma, \Sigma) \models \lVert p \rVert$ iff $\exists w, \mathbb{D}. (\sigma, w, \mathbb{D}, \Sigma) \models p$

$\mathbb{D}_1 \perp \mathbb{D}_2$ iff $(\mathbb{D}_1 = \bullet) \vee (\mathbb{D}_2 = \bullet)$ $\qquad \mathbb{D}_1 \uplus \mathbb{D}_2 \stackrel{\text{def}}{=} \begin{cases} \mathbb{D}_2 & \text{if } \mathbb{D}_1 = \bullet \\ \mathbb{D}_1 & \text{if } \mathbb{D}_2 = \bullet \end{cases}$

$(\sigma_1, w_1, \mathbb{D}_1, \Sigma_1) \uplus (\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \stackrel{\text{def}}{=}$
$(\sigma_1 \uplus \sigma_2, w_1 + w_2, \mathbb{D}_1 \uplus \mathbb{D}_2, \Sigma_1 \uplus \Sigma_2)$, if $\sigma_1 \perp \sigma_2, \mathbb{D}_1 \perp \mathbb{D}_2$ and $\Sigma_1 \perp \Sigma_2$

$\mathsf{Sta}(p, R)$ iff $\forall \sigma, w, \mathbb{D}, \Sigma, \sigma', \Sigma', b.$
$((\sigma, w, \mathbb{D}, \Sigma) \models p) \wedge (((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R)$
$\implies \exists w'. (\sigma, w', \mathbb{D}, \Sigma') \models p \ \wedge\ (b = \textbf{false} \implies w' = w)$

$p \Rrightarrow^0 q$ iff $p \Rightarrow q$
$p \Rrightarrow^+ q$ iff $\forall \sigma, w, \mathbb{D}, \Sigma, \Sigma_F. ((\sigma, w, \mathbb{D}, \Sigma) \models p) \wedge (\Sigma \perp \Sigma_F) \implies$
$\exists w', \mathbb{C}', \Sigma'. (\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F) \wedge ((\sigma, w', \mathbb{C}', \Sigma') \models q)$

Figure 8. Semantics of assertions (part II).

5.1 Assertions on Source Code and Number of Tokens

We first explain the new assertions p and q used in the unary rules that can specify the source code and metrics in addition to states. We define their syntax in Fig. 5, and their semantics in Fig. 8. A *full state assertion* p is interpreted on $(\sigma, w, \mathbb{D}, \Sigma)$. Here besides the states σ and Σ at the target and source levels, we introduce some auxiliary data w and $\mathbb{D}$. w is the number of tokens needed for loops (see Sec. 2). $\mathbb{D}$ is either some source code $\mathbb{C}$, or a special sign $\bullet$ serving as a unit for defining semantics of $p * q$ below.

In Fig. 8 we lift the relational assertion P as a full state assertion to specify the states. The new assertion $\mathsf{arem}(\mathbb{C})$ says that the remaining source code is $\mathbb{C}$ at the current program point. $\mathsf{wf}(E)$ states that the number of tokens at the current target code is *no less than* E. We can see $\mathsf{wf}(0)$ always holds, and for any n, $\mathsf{wf}(n+1)$ implies $\mathsf{wf}(n)$. We use $\lfloor p \rfloor_{\mathsf{a}}$ and $\lfloor p \rfloor_{\mathsf{w}}$ to ignore the descriptions in p about the source code and the number of tokens respectively. $\lVert p \rVert$ lifts p back to a relational state assertion.

Separating conjunction $p * q$ has the standard meaning as in separation logic, which says p and q hold over disjoint parts of $(\sigma, w, \mathbb{D}, \Sigma)$ respectively (the formal definition elided here). However, it is worth noting the definition of disjoint union over the quadruple states, which is shown in the middle part of Fig. 8. The disjoint union of the numbers of tokens w_1 and w_2 is simply the sum of them. The disjoint union of $\mathbb{D}_1$ and $\mathbb{D}_2$ is defined only if

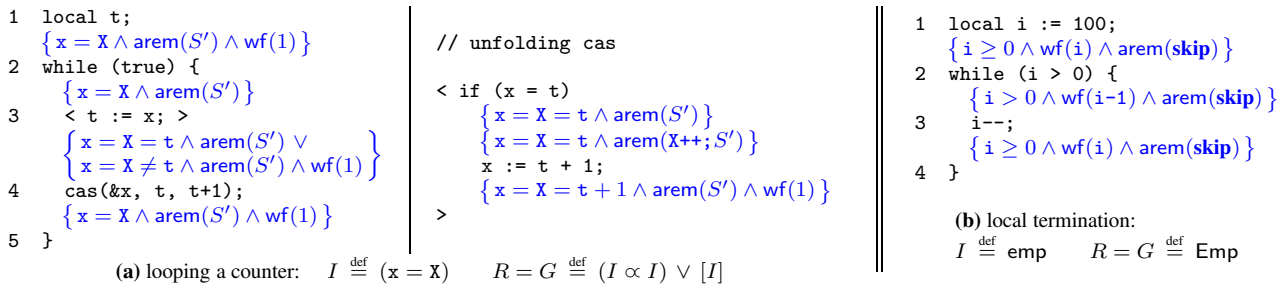
$$\frac{R, G, I \vdash \{P \wedge \mathsf{arem}(\mathbb{C})\} C \{Q \wedge \mathsf{arem}(\mathbf{skip})\}}{R, G, I \vdash \{P\} C \preceq \mathbb{C} \{Q\}} \ (\text{U2B})$$

$$\frac{\vdash_{\text{SL}} [p] C [q] \quad (\llbracket p \rrbracket \ltimes \llbracket q \rrbracket) \Rightarrow G * \mathsf{True} \quad I \triangleright G \quad p \vee q \Rightarrow I * \mathbf{true}}{[I], G, I \vdash \{p\}\langle C \rangle \{q\}} \ (\text{ATOM}) \qquad \frac{p \Rrightarrow^{a} p' \quad \vdash_{\text{SL}} [p'] C [q'] \quad q' \Rrightarrow^{b} q \quad + \in \{a, b\} \quad (\llbracket p \rrbracket \propto \llbracket q \rrbracket) \Rightarrow G * \mathsf{True} \quad I \triangleright G \quad p \vee q \Rightarrow I * \mathbf{true}}{[I], G, I \vdash \{p\}\langle C \rangle \{q\}} \ (\text{ATOM}^{+})$$

$$\frac{[I], G, I \vdash \{p\}\langle C \rangle \{q\} \quad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \quad I \triangleright R}{R, G, I \vdash \{p\}\langle C \rangle \{q\}} \ (\text{ATOM-R}) \qquad \frac{p \Rightarrow (B = B) * I \quad p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp}) \quad R, G, I \vdash \{p'\} C \{p\}}{R, G, I \vdash \{p\} \mathbf{while}\ (B)\ C \{p \wedge \neg B\}} \ (\text{WHILE})$$

$$\frac{R, G, I \vdash \{p\} C \{q\}}{R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{w}}\} C \{\lfloor q \rfloor_{\mathsf{w}}\}} \ (\text{HIDE-W}) \qquad \frac{R, G, I \vdash \{p\} C \{q\} \quad \mathsf{Sta}(p', R' * \mathsf{Id}) \quad I' \triangleright \{R', G'\} \quad p' \Rightarrow I' * \mathbf{true} \quad G^{+} \Rightarrow G}{R * R', G * G', I * I' \vdash \{p * p'\} C \{q * p'\}} \ (\text{FRAME})$$

Figure 9. Selected unary inference rules.



(a) looping a counter: $I \stackrel{\text{def}}{=} (\mathtt{x} = \mathtt{X}) \qquad R = G \stackrel{\text{def}}{=} (I \propto I) \vee [I]$

(b) local termination: $I \stackrel{\text{def}}{=} \mathsf{emp} \qquad R = G \stackrel{\text{def}}{=} \mathsf{Emp}$

Figure 10. Proofs for two small examples.

at least one of them is the special sign $\bullet$, which has no knowledge about the remaining source code $\mathbb{C}$. Therefore we know the following holds (for any P and $\mathbb{C}$):

$$(P \wedge \mathsf{arem}(\mathbb{C}) \wedge \mathsf{wf}(1)) * (\mathsf{wf}(1) \wedge \mathsf{emp}) \ \Leftrightarrow \ (P \wedge \mathsf{arem}(\mathbb{C}) \wedge \mathsf{wf}(2))$$

One may think a more natural definition of the disjoint union is to require the two $\mathbb{D}$s be the same. But this would break the FRAME rule (see Fig. 9). For example, we can prove:

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{\mathtt{x} = \mathtt{X} \wedge \mathsf{arem}(\mathtt{X{+}{+}})\}\ \mathtt{x{+}{+}}\ \{\mathtt{x} = \mathtt{X} \wedge \mathsf{arem}(\mathbf{skip})\}$$

With the FRAME rule and the separating conjunction based on the alternative definition of disjoint union, we would derive the following:

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{(\mathtt{x} = \mathtt{X} \wedge \mathsf{arem}(\mathtt{X{+}{+}})) * \mathsf{arem}(\mathtt{X{+}{+}})\}\ \mathtt{x{+}{+}}\ \{(\mathtt{x} = \mathtt{X} \wedge \mathsf{arem}(\mathbf{skip})) * \mathsf{arem}(\mathtt{X{+}{+}})\}$$

which is reduced to an invalid judgment:

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{\mathtt{x} = \mathtt{X} \wedge \mathsf{arem}(\mathtt{X{+}{+}})\}\ \mathtt{x{+}{+}}\ \{\mathbf{false}\}$$

We require in $p * q$ that either p or q should *not* specify the source code, therefore in this example the precondition after applying the frame rule is invalid (thus the whole judgment is valid).

The stability of p w.r.t. an action R, defined at the bottom part of Fig. 8, specifies how the number of tokens of a program (specified by p) could change under R's interferences. As a simple example, for the following p, R_1 and R_2, $\mathsf{Sta}(p, R_1)$ holds while $\mathsf{Sta}(p, R_2)$ does not hold:

$$p \stackrel{\text{def}}{=} (10 \mapsto 0 * 20 \Rrightarrow 0) \vee ((10 \mapsto 1 * 20 \Rrightarrow 0) \wedge \mathsf{wf}(1))$$
$$R_1 \stackrel{\text{def}}{=} (10 \mapsto 0 * 20 \Rrightarrow 0) \propto (10 \mapsto 1 * 20 \Rrightarrow 0)$$
$$R_2 \stackrel{\text{def}}{=} (10 \mapsto 0 * 20 \Rrightarrow 0) \ltimes (10 \mapsto 1 * 20 \Rrightarrow 0)$$

5.2 Unary Inference Rules

The judgment for unary reasoning is in the form of $R, G, I \vdash \{p\} C \{q\}$. We present some of the rules in Fig. 9.

The U2B rule, as explained in Sec. 2, turns unary proofs to binary ones. It says that if the remaining source code is $\mathbb{C}$ at the beginning of the target C, and it becomes **skip** at the end of C, then we know C is simulated by $\mathbb{C}$.

The ATOM rule allows us to reason sequentially about the target code in the atomic block. We use $\vdash_{\text{SL}} [p] C [q]$ to represent the total correctness of C in sequential separation logic. The corresponding rules are mostly standard and elided here. Note that C only accesses the target state σ, therefore in our sequential rules we require the source state Σ and the auxiliary data w and $\mathbb{D}$ in p should remain unchanged in q. We can lift C's total correctness to the concurrent setting as long as its overall transition over the shared states satisfies the guarantee G. Here we assume the environment is identity transitions. To allow general environment behaviors, we can apply the ATOM-R rule later, which requires that R be fenced by I and the pre- and post-conditions be stable w.r.t. R.

The ATOM^{+} rule is similar to the ATOM rule, except that it executes the source code simultaneously with the target atomic step. We use $p \Rrightarrow^{+} q$ for the multi-step executions from the source code specified by p to the code specified by q, which is defined in the bottom part of Fig. 8. We also write $p \Rrightarrow^{0} q$ for the usual implication $p \Rightarrow q$. Then, the ATOM^{+} rule says, we can execute the source code before or after the steps of C, as long as the overall transition (including the source steps *and* the target steps) with the effect bit **true** satisfies G for the shared parts.

The WHILE rule is the key to proving the preservation of termination. As we informally explained in Sec. 2, we should be able to decrease the number of tokens at the beginning of each loop iteration. And we should re-establish the invariant p between the states and the number of tokens at the end of each iteration. Below we give two examples, each of which shows a typical application of the WHILE rule.

Examples. The first example is the T_c'' and S' in Sec. 2. We show its proof in our logic in Fig. 10(a) (for simplicity, below we always assume the ownership of variables). We use `X` for the counter at the source, and the rely/guarantee conditions say that the counters at the two levels can be updated simultaneously with the effect bit **true**. The loop invariant above line 2 says that we should have at least one token to execute the loop. The loop body is verified with zero tokens, and should finally restore the invariant token number 1. The gaining of the token may be due to a successful `cas` at line 4 that corresponds to source steps, or caused by the environment interferences. More specifically, the assertion following line 3 says that we can gain a token if the counters have been updated. If the counters are not updated before the `cas` at line 4, the `cas` succeeds and we show the detailed proof at the right part of Fig. 10(a), in which we execute one iteration of the source code and gain a token (applying the ATOM$^+$ rule).

This example shows the most straightforward understanding of the WHILE rule: we pay a token at the beginning of an iteration and should be able to gain another token during the execution of the iteration. The next example is more subtle (though simpler). As shown in Fig. 10(b), it is a locally-terminating while loop (i.e., a loop that terminates regardless of environment interferences). We prove it refines **skip** under the environment `Emp`. The loop invariant above line 2 says that the number of tokens equals the value of `i`. If the loop condition (`i>0`) is satisfied, we pay one token. In the proof of the loop body, we do not (and are not able to) gain more tokens. Instead, the value of `i` will be decreased in the iteration, enabling us to restore the equality between the number of tokens and `i`.

Other rules and discussions. Another important rule is the HIDE-W rule in Fig. 9. It shows that tokens are just an auxiliary tool, which could be safely discarded (by using $\lfloor\_\rfloor_w$) when the termination-preservation of a command C (say, a while loop) is already established. As we mentioned in Sec. 2, the HIDE-W rule is crucial to handle *infinite nondeterminism*. It is also important for local reasoning, so that when we verify a thread, we do not have to calculate and specify in the precondition the number of tokens needed by *all* the while loops. For nested loops, we could use the HIDE-W rule to hide the tokens needed by the inner loop, and use the FRAME rule to add back the tokens needed for the outer loop later when we compose the inner loop with other parts of the outer loop body.

The unary FRAME rule in Fig. 9 is similar to the binary one in Fig. 7. Other rules can be found in our TR [13], which are very similar to those in LRG [3], but we give different interpretations to assertions and actions.

The binary rules (in Fig. 7) and the unary rules (in Fig. 9) gives us a full proof theory for termination-preserving refinement. We want to remind the readers that the logic does not ensure termination of programs, therefore it is *not* a logic for total correctness. On the other hand, if we restrict the source code to **skip** (which always terminates), then our unary rules can be viewed as a proof theory for the *total* correctness of concurrent programs.

Also note that the use of a natural number w as the while-specific metric is to simplify the presentation only. It is easy to extend our work to support other types of the while-specific metrics for more complicated examples.

6. More Examples

We have seen a few small examples that illustrate the use of our logic, in particular, the WHILE rule. In this section, we discuss other examples that we have proved, which are summarized in Fig. 11. Their proofs are in TR [13].

Linearizability & Lock-Freedom	Counter and its variants
	Treiber stack [20]
	Michael-Scott lock-free queue [14]
	DGLM lock-free queue [2]
Non-Atomic Object Correctness	Synchronous queue [16]
Correctness of Optimized Algo (Equivalence)	Counter vs. its variants
	TAS lock vs. TTAS lock [6]

Figure 11. Verified examples using our logic.

Proving linearizability and lock-freedom together for concurrent objects. It has been shown [12] that the verification of linearizability and lock-freedom together can be reduced to verifying a contextual refinement that preserves the termination of any client programs. That is, for any client as the context $\mathscr{C}$, the termination-preserving refinement $\mathscr{C}[C] \sqsubseteq \mathscr{C}[\mathbb{C}]$ should hold. Here we use C for the concrete implementation of the object, and $\mathbb{C}$ for the corresponding abstract atomic operations. $\mathscr{C}[C]$ (or $\mathscr{C}[\mathbb{C}]$) denotes the whole program where the client accesses the object via method calls to C (or $\mathbb{C}$).

The compositionality rules of our logic (Fig. 7) allow us to verify the above contextual refinement by proving $R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}$. Then we apply the U2B rule and turn the relational verification to unary reasoning. As in a normal linearizability proof (e.g., [10, 23]), we need to find a single step of C (i.e., the linearization point) that corresponds to the atomic step of $\mathbb{C}$. Here we also have to prove lock-freedom: the failure to make progress (i.e., finish an abstract operation) of a thread must be caused by successful progress of its environment, which can be ensured by the WHILE rule (in Fig. 9) in our logic.

We have used the above approach to verify several linearizable and lock-free objects, including Treiber stack [20], Michael-Scott lock-free queue [14] and DGLM queue [2]. We can further extend the logic in this paper with the techniques [10] for verifying linearizability of algorithms with non-fixed linearization points, to support more sophisticated examples such as HSY elimination-based stack and Harris-Michael lock-free list.

Verifying concurrent objects whose abstract operations are* not *atomic. Sometimes we cannot define single atomic operations as the abstract specification of a concurrent object. For objects that implement synchronization between threads, we may have to explicitly take into account the interferences from other threads when defining the abstract behaviors of the current thread. For example, the synchronous queue [16] is a concurrent transfer channel in which each producer presenting an item must wait for a consumer to take this item, and vice versa. The corresponding abstract operations are no longer atomic. We used our logic to prove the contextual refinement between the concrete implementation (from [16], used in Java 6) and a more abstract synchronous queue. The refinement ensures that if a producer (or a consumer) is blocked at the concrete level, it must also be blocked at the source level.

Proving equivalence between optimized algorithms and original ones. We also use our logic to show variants of concurrent algorithms are correct optimizations of the original implementations. In this case, we show equivalence (in fact, contextual equivalence), i.e., refinements of both directions.

For instance, we proved the TTAS lock implementation is equivalent to the TAS lock implementation [6] for any client using the locks. The former tests the lock bit in a nested while loop until it appears to be free, and then uses the atomic `getAndSet` instruction to update the bit; while the latter directly tries `getAndSet` until success. The equivalence result between these two lock implementations shows that no client may observe their differences, including the differences on their termination behaviors (e.g., whether a

client thread may acquire the lock). It gives us the full correctness of the TTAS lock. As an optimization of TAS lock, it preserves the behaviors on both functionality and termination of the latter.

7. Related Work and Conclusion

Hoffmann et al. [7] propose a program logic to verify lock-freedom of concurrent objects. They reason about termination quantitatively by introducing tokens, and model the environment's interference over the current thread's termination in terms of token transfer. The idea is simple and natural, but their logic has very limited support of local reasoning. One needs to know the total number of tokens needed by each thread (which may have multiple while loops) and the (fixed) number of threads, to calculate the number of tokens for a thread to lose or initially own. This requirement also disallows their logic to reason about programs with infinite nondeterminism. Here we allow a thread to set its effect bit in R/G without knowing the details of other threads; and other threads can determine by themselves how many tokens they gain. We also introduce the HIDE-W rule to hide the number of tokens and to support infinite nondeterminism. Another key difference is that our logic supports verification of refinement, which is not supported by their logic.

Gotsman et al. [5] propose program logic and tools to verify lock-freedom. Their approach is more heavyweight in that they need temporal assertions in the rely/guarantee conditions to specify interference between threads, and the rely/guarantee conditions need to be specified iteratively in multiple rounds to break circular reliance on progress. Moreover, their work relies on third-party tools to check termination of individual threads as closed sequential programs. Therefore they do not have a set of self-contained program logic rules and a coherent meta-theory as we do. Like Hoffmann et al. [7], they do not support refinement verification either.

As we explained in Sec. 1, none of recent work on general refinement verification of concurrent programs [11, 21, 22] and on verifying linearizability of concurrent objects [10, 23] (which can be viewed as a specialized refinement problem) preserves termination. Ševčík et al. equipped their simulation proofs for CompCertTSO [17] with a well-founded order, following the CompCert approach. Their approach is similar to our second attempt explained in Sec. 2, thus cannot be applied to prove lock-freedom of concurrent objects.

Conclusion and future work. We propose a new compositional simulation RGSim-T to verify termination-preserving refinement between concurrent programs. We also give a rely/guarantee program logic as a proof theory for the simulation. Our logic is the first to support compositional verification of termination-preserving refinement. The simulation and logic are general. They can be used to verify both correctness of optimizations (where the source may not necessarily terminate) and lock-freedom of concurrent objects. As future work, we would like to further extend them with the techniques of pending thread pools and speculations [10] to verify objects with non-fixed linearization points. We also hope to explore the possibility of building tools to automate the verification.

Acknowledgments

We thank anonymous referees for their suggestions and comments. This work is supported in part by China Scholarship Council, National Natural Science Foundation of China (NSFC) under Grant Nos. 61229201, 61379039 and 91318301, and the National Hi-Tech Research and Development Program of China (Grant No. 2012AA010901). It is also supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0915888 and 1065451. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

[2] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, pages 97–114, 2004.

[3] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327, 2009.

[4] I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[5] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28, 2009.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[7] J. Hoffmann, M. Marmar, and Z. Shao. Quantitative reasoning for proving lock-freedom. In *LICS*, pages 124–133, 2013.

[8] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[9] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43:363–446, December 2009.

[10] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[11] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, pages 455–468, 2012.

[12] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, pages 227–241, 2013.

[13] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs (extended version). Technical report, Univ. of Science and Technology of China, May 2014. `http://kyhcs.ustcsz.edu.cn/relconcur/rgsimt`.

[14] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[15] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, pages 137–146, 2006.

[16] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP*, pages 147–156, 2006.

[17] J. Ševčík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22, 2013.

[18] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. In *PLDI*, pages 471–482, 2013.

[19] K. Stølen. A method for the development of totally correct shared-state parallel programs. In *CONCUR*, pages 510–525, 1991.

[20] R. K. Treiber. System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

[21] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013.

[22] A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, pages 343–356, 2013.

[23] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, Computer Laboratory, 2008.

[24] V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, pages 335–351, 2011.

[25] H. Yang. Relational separation logic. *Theoretical Computer Science*, 375:308–334, 2007.

Compositional Verification of Termination-Preserving Refinement of Concurrent Programs (Technical Report)

Hongjin Liang[1], Xinyu Feng[1], and Zhong Shao[2]

[1]University of Science and Technology of China
[2]Yale University

May 26, 2015

NOTES: This TR is a supplement to our CSL-LICS'14 paper. It includes full formulations of the technical settings (Section 1), our RGSim-T definitions (Section 2), the full program logic (Section 3), all the examples we have verified (Section 4) and the full formal soundness proofs (Section 5).

Moreover, we introduce a new interesting assertion $p \oslash q$ which allows local reasoning about the number of tokens that is conditional upon the shared state in runtime. See Section 2 for its semantics, Section 3 for the related local reasoning rule and Section 4 for its use in practical examples.

We also provide a transitivity rule on the binary judgments. We introduce new assertions to specify the compositions of two relational assertions and of two actions (see Section 2).

For more informal explanations and the high-level picture, please see our CSL-LICS'14 paper. Both the paper and this companion TR can be found at the following url:

`http://kyhcs.ustcsz.edu.cn/relconcur/rgsimt`

1 Basic Technical Settings and Termination-Preserving Refinement

1.1 The Language

We show the language in Figure 1. We assume the program variables used in the target code are different from the ones used in the source (e.g., we use x and X for target and source level variables respectively).

$$
\begin{array}{llllll}
(\mathit{Event}) & e & ::= & \ldots & (\mathit{Label}) \quad \iota ::= e \mid \tau \\
(\mathit{Store}) & s, \mathbb{s} & \in & \mathit{PVar} \rightharpoonup \mathit{Val} & (\mathit{Heap}) \quad h, \mathbb{h} \in \mathit{Addr} \rightharpoonup \mathit{Val} \\
(\mathit{State}) & \sigma, \Sigma & ::= & (s, h) & \\
(\mathit{Instr}) & c, \mathbb{c} & \in & \mathit{State} \rightharpoonup \mathscr{P}((\mathit{Label} \times \mathit{State}) \cup \{\mathbf{abort}\}) & \\
(\mathit{Expr}) & E, \mathbb{E} & ::= & x \mid n \mid E + E \mid \ldots & \\
(\mathit{BExp}) & B, \mathbb{B} & ::= & \mathbf{true} \mid \mathbf{false} \mid E = E \mid !B \mid \ldots & \\
(\mathit{Stmt}) & C, \mathbb{C} & ::= & \mathbf{skip} \mid c \mid \langle C \rangle \mid C_1; C_2 \mid \mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2 & \\
 & & \mid & \mathbf{while}\ (B)\ C \mid C_1 \,\|\, C_2 &
\end{array}
$$

Figure 1: Generic language at target and source levels.

We show the operational semantics in Figure 2. The semantics of E and B are defined by $[\![E]\!]$ and $[\![B]\!]$ respectively. $[\![E]\!]$ is a partial function of type $\mathit{Store} \rightharpoonup \mathit{Val}$. $[\![B]\!]$ is a partial function of type $\mathit{Store} \rightharpoonup \{\mathbf{true}, \mathbf{false}\}$. They are undefined if variables in E and B are not assigned values in the store s. Their definitions are omitted here.

Conventions. We usually write blackboard bold or capital letters ($\mathbb{s}$, $\mathbb{h}$, Σ, $\mathbb{c}$, $\mathbb{E}$, $\mathbb{B}$ and $\mathbb{C}$) for the notations at the source level to distinguish from the target-level ones (s, h, σ, c, E, B and C). When we discuss the transitivity, we use θ and $\mathrm{C_M}$ for the state and the code at the middle level.

Below we use $\_ \longrightarrow^* \_$ for zero or multiple-step transitions with no events generated, $\_ \longrightarrow^+ \_$ for multiple-step transitions without events, $\_ \xrightarrow{e}{}^+ \_$ for multiple-step transitions with *only one* event e generated, and $\_ \longrightarrow^\omega \cdot$ for an infinite execution without events.

$$\frac{(\iota, \sigma') \in c\ \sigma}{(c, \sigma) \xrightarrow{\iota} (\mathbf{skip}, \sigma')} \qquad \frac{\mathbf{abort} \in c\ \sigma}{(c, \sigma) \longrightarrow \mathbf{abort}} \qquad \frac{\sigma \notin dom(c)}{(c, \sigma) \longrightarrow (c, \sigma)}$$

$$\frac{(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')}{(\langle C \rangle, \sigma) \longrightarrow (\mathbf{skip}, \sigma')} \qquad \frac{(C, \sigma) \longrightarrow^* \mathbf{abort}}{(\langle C \rangle, \sigma) \longrightarrow \mathbf{abort}} \qquad \frac{(C, \sigma) \longrightarrow^\omega \cdot}{(\langle C \rangle, \sigma) \longrightarrow (\langle C \rangle, \sigma)}$$

$$\frac{(C, \sigma) \longrightarrow (C', \sigma')}{(C; C'', \sigma) \longrightarrow (C'; C'', \sigma')} \qquad \frac{(C, \sigma) \xrightarrow{e} (C', \sigma')}{(C; C'', \sigma) \xrightarrow{e} (C'; C'', \sigma')}$$

$$\frac{}{(\mathbf{skip}; C', \sigma) \longrightarrow (C', \sigma)} \qquad \frac{(C, \sigma) \longrightarrow \mathbf{abort}}{(C; C', \sigma) \longrightarrow \mathbf{abort}}$$

$$\frac{[\![B]\!]_s = \mathbf{true}}{(\mathbf{while}\ (B)\ C, (s, h)) \longrightarrow (C; \mathbf{while}\ (B)\ C, (s, h))}$$

$$\frac{[\![B]\!]_s = \mathbf{false}}{(\mathbf{while}\ (B)\ C, (s, h)) \longrightarrow (\mathbf{skip}, (s, h))} \qquad \frac{[\![B]\!]_s\ \textit{undefined}}{(\mathbf{while}\ (B)\ C, (s, h)) \longrightarrow \mathbf{abort}}$$

$$\frac{[\![B]\!]_s = \mathbf{true}}{(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, (s, h)) \longrightarrow (C_1, (s, h))} \qquad \frac{[\![B]\!]_s = \mathbf{false}}{(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, (s, h)) \longrightarrow (C_2, (s, h))}$$

$$\frac{[\![B]\!]_s\ \textit{undefined}}{(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, (s, h)) \longrightarrow \mathbf{abort}}$$

$$\frac{(C_1, \sigma) \xrightarrow{\iota} (C_1', \sigma')}{(C_1 \,\|\, C_2, \sigma) \xrightarrow{\iota} (C_1' \,\|\, C_2, \sigma')} \qquad \frac{(C_2, \sigma) \xrightarrow{\iota} (C_2', \sigma')}{(C_1 \,\|\, C_2, \sigma) \xrightarrow{\iota} (C_1 \,\|\, C_2', \sigma')}$$

$$\frac{}{(\mathbf{skip} \,\|\, \mathbf{skip}, \sigma) \longrightarrow (\mathbf{skip}, \sigma)} \qquad \frac{(C_1, \sigma) \longrightarrow \mathbf{abort} \quad \text{or} \quad (C_2, \sigma) \longrightarrow \mathbf{abort}}{(C_1 \,\|\, C_2, \sigma) \longrightarrow \mathbf{abort}}$$

Figure 2: Operational semantics.

1.2 Termination-Preserving Event Trace Refinement

$$(\mathit{EvtTrace}) \quad \mathcal{E} \quad ::= \quad \Downarrow \mid \lightning \mid \epsilon \mid e :: \mathcal{E} \quad \text{(co-inductive interpretation)}$$

We define $ETr(C, \sigma, \mathcal{E})$ in Figure 3.[1]

$$\frac{(C,\sigma) \longrightarrow^* (\mathbf{skip}, \sigma')}{ETr(C,\sigma,\Downarrow)} \qquad \frac{(C,\sigma) \longrightarrow^+ \mathbf{abort}}{ETr(C,\sigma,\lightning)}$$

$$\frac{(C,\sigma) \longrightarrow^+ (C',\sigma') \qquad ETr(C',\sigma',\epsilon)}{ETr(C,\sigma,\epsilon)} \qquad \frac{(C,\sigma) \xrightarrow{e}{}^+ (C',\sigma') \qquad ETr(C',\sigma',\mathcal{E})}{ETr(C,\sigma,e::\mathcal{E})}$$

Figure 3: Co-inductive definition of $ETr(C, \sigma, \mathcal{E})$.

Definition 1 (Termination-Preserving Refinement).
$(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$ iff $\forall \mathcal{E}.\ ETr(C, \sigma, \mathcal{E}) \implies ETr(\mathbb{C}, \Sigma, \mathcal{E})$.

[1] We made a typo in the definition of ETr in our published paper. In the paper, the third rule is as follows.

$$\frac{(C,\sigma) \longrightarrow^+ (C',\sigma') \qquad ETr(C',\sigma',\mathcal{E})}{ETr(C,\sigma,\mathcal{E})}$$

Such a definition is incorrect because it allows any event trace to be an acceptable trace of **while** (**true**){**skip**}. We corrected it by restricting the trace of an infinite loop to be empty, as shown in Figure 3.

2 RGSim-T

2.1 Assertion Language

We first define the assertions used in our simulation RGSim-T and our program logic. Their syntax is shown in Figure 4, and their semantics is shown in Figures 5 and 6.

$$
\begin{array}{rrcl}
(\mathit{RelAssn}) & P, Q, I & ::= & B \mid \mathsf{own}(x) \mid \mathit{emp} \mid \boldsymbol{emp} \mid E \mapsto E \mid E \Mapsto E \\
& & & \mid [\![p]\!] \mid P * Q \mid P \vee Q \mid P \wedge Q \mid P \fatsemi Q \mid \ldots \\
(\mathit{FullAssn}) & p, q & ::= & P \mid \mathsf{arem}(\mathbb{C}) \mid \mathsf{wf}(E) \mid \lfloor p \rfloor_{\mathsf{a}} \mid \lfloor p \rfloor_{\mathsf{w}} \\
& & & \mid p * q \mid p \vee q \mid p \wedge q \mid p \mathbin{\textcircled{\&}} q \mid \ldots \\
(\mathit{RelAct}) & R, G & ::= & P \propto Q \mid P \ltimes Q \mid [P] \mid R * R \mid R^{+} \\
& & & \mid R \vee R \mid R \wedge R \mid R \mathbin{\hat{\fatsemi}} R \mid R \mathbin{\check{\fatsemi}} R \mid \ldots
\end{array}
$$

Figure 4: Assertion language.

The above assertion language extends the one in our CSL-LICS paper with the following new assertions.

1. $p \mathbin{\textcircled{\&}} q$, which is like a conjunction over the concrete and the abstract states and like a separating conjunction over the number of tokens and the abstract code. It would be useful to simplify the verification of some specific examples (see Section 4).

2. $P \fatsemi Q$, $R \mathbin{\hat{\fatsemi}} R$ and $R \mathbin{\check{\fatsemi}} R$, which are compositions of two relational assertions and of two actions. They are used in the transitivity of the binary judgments (the TRANS rule in Figure 7). We use θ and $\mathrm{C_M}$ to represent the middle-level state and the middle-level code respectively. We also define a predicate $\mathsf{MPrecise}(P, Q)$ in Figure 5, which specifies the precise property about the middle-level states. Here P and Q are relational assertions between low-level and middle-level states and between middle-level and high-level states respectively.

Note that our logic is already very useful without the above extensions. All the examples that we mentioned in our CSL-LICS'14 paper can be verified without these extensions.

$f_1 \bot f_2$ iff $(dom(f_1) \cap dom(f_2) = \emptyset)$

$(s_1, h_1) \bot (s_2, h_2)$ iff $(s_1 \bot s_2) \wedge (h_1 \bot h_2)$

$$(s_1, h_1) \uplus (s_2, h_2) \stackrel{\text{def}}{=} \begin{cases} (s_1 \cup s_2, h_1 \cup h_2) & \text{if } (s_1, h_1) \bot (s_2, h_2) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

$((s, h), (\mathbb{s}, \mathbb{h})) \models B$ iff $[\![B]\!]_{s \uplus \mathbb{s}} = \mathbf{true}$

$((s, h), (\mathbb{s}, \mathbb{h})) \models \mathsf{own}(x)$ iff $dom(s \uplus \mathbb{s}) = \{x\}$

$((s, h), (\mathbb{s}, \mathbb{h})) \models emp$ iff $(dom(s) = \emptyset) \wedge (dom(h) = \emptyset)$

$((s, h), (\mathbb{s}, \mathbb{h})) \models \boldsymbol{emp}$ iff $(dom(\mathbb{s}) = \emptyset) \wedge (dom(\mathbb{h}) = \emptyset)$

$((s, h), (\mathbb{s}, \mathbb{h})) \models E_1 \mapsto E_2$ iff $\exists l, n.\ [\![E_1]\!]_{s \uplus \mathbb{s}} = l \wedge [\![E_2]\!]_{s \uplus \mathbb{s}} = n \wedge dom(h) = \{l\} \wedge h(l) = n$

$((s, h), (\mathbb{s}, \mathbb{h})) \models E_1 \mapsto\!\!\!\!\Rightarrow E_2$ iff $\exists l, n.\ [\![E_1]\!]_{s \uplus \mathbb{s}} = l \wedge [\![E_2]\!]_{s \uplus \mathbb{s}} = n \wedge dom(\mathbb{h}) = \{l\} \wedge \mathbb{h}(l) = n$

$\mathsf{emp} \stackrel{\text{def}}{=} emp \wedge \boldsymbol{emp}$

$(\sigma, \Sigma) \models P \fatsemi Q$ iff $\exists \theta.\ (\sigma, \theta) \models P \wedge (\theta, \Sigma) \models Q$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models P \propto Q$ iff $(\sigma, \Sigma) \models P \wedge (\sigma', \Sigma') \models Q \wedge (b = \mathbf{true})$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models P \ltimes Q$ iff $(\sigma, \Sigma) \models P \wedge (\sigma', \Sigma') \models Q$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models [P]$ iff $(\sigma, \Sigma) \models P \wedge (\sigma = \sigma') \wedge (\Sigma = \Sigma')$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R_1 * R_2$ iff
$\exists \sigma_1, \Sigma_1, \sigma_2, \Sigma_2, \sigma_1', \Sigma_1', \sigma_2', \Sigma_2'.\ ((\sigma_1, \Sigma_1), (\sigma_1', \Sigma_1'), b) \models R_1 \wedge ((\sigma_2, \Sigma_2), (\sigma_2', \Sigma_2'), b) \models R_2$
$\wedge\ (\sigma = \sigma_1 \uplus \sigma_2) \wedge (\sigma' = \sigma_1' \uplus \sigma_2') \wedge (\Sigma = \Sigma_1 \uplus \Sigma_2) \wedge (\Sigma' = \Sigma_1' \uplus \Sigma_2')$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R^+$ iff
$(((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R)$
$\vee\ (\exists \sigma'', \Sigma'', b', b''.\ (((\sigma, \Sigma), (\sigma'', \Sigma''), b') \models R) \wedge (((\sigma'', \Sigma''), (\sigma', \Sigma'), b'') \models R^+) \wedge (b = b' \vee b''))$

$\mathsf{Id} \stackrel{\text{def}}{=} [\mathbf{true}]$ $\quad \mathsf{Emp} \stackrel{\text{def}}{=} \mathsf{emp} \ltimes \mathsf{emp}$ $\quad \mathsf{True} \stackrel{\text{def}}{=} \mathbf{true} \ltimes \mathbf{true}$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R_1 \mathbin{\hat{\fatsemi}} R_2$ iff
$\exists \theta, \theta', b_1, b_2.\ ((\sigma, \theta), (\sigma', \theta'), b_1) \models R_1 \wedge ((\theta, \Sigma), (\theta', \Sigma'), b_2) \models R_2 \wedge (b = b_1 \wedge b_2)$

$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R_1 \mathbin{\check{\fatsemi}} R_2$ iff
$\exists \theta, \theta', b_1, b_2.\ ((\sigma, \theta), (\sigma', \theta'), b_1) \models R_1 \wedge ((\theta, \Sigma), (\theta', \Sigma'), b_2) \models R_2 \wedge (b = b_1 \vee b_2)$

$\mathsf{Sta}(P, R)$ iff $\forall \sigma, \Sigma, \sigma', \Sigma', b.\ ((\sigma, \Sigma) \models P) \wedge (((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R) \implies ((\sigma', \Sigma') \models P)$

$\mathsf{Precise}(P)$ iff $\forall \sigma_1, \Sigma_1, \sigma_2, \Sigma_2, \sigma_1', \Sigma_1', \sigma_2', \Sigma_2'.$
$((\sigma_1 \uplus \sigma_2 = \sigma_1' \uplus \sigma_2') \wedge ((\sigma_1, \_) \models P) \wedge ((\sigma_1', \_) \models P) \implies (\sigma_1 = \sigma_1'))$
$\wedge\ ((\Sigma_1 \uplus \Sigma_2 = \Sigma_1' \uplus \Sigma_2') \wedge ((\_, \Sigma_1) \models P) \wedge ((\_, \Sigma_1') \models P) \implies (\Sigma_1 = \Sigma_1'))$

$I \triangleright R$ iff $([I] \Rightarrow R) \wedge (R \Rightarrow I \ltimes I) \wedge \mathsf{Precise}(I)$

$\mathsf{MPrecise}(P, Q)$ iff
$\forall \theta_1, \theta_1', \theta_2, \theta_2'.\ (\theta_1 \uplus \theta_2 = \theta_1' \uplus \theta_2') \wedge ((\_, \theta_1) \models P) \wedge ((\theta_1', \_) \models Q) \implies (\theta_1 = \theta_1')$

Figure 5: Semantics of assertions (part I).

$$
\begin{array}{lll}
(\mathit{HCState}) & \mathbb{D} & ::= \quad \mathbb{C} \mid \bullet \\
(\mathit{FullState}) & \mathcal{S} & ::= \quad (\sigma, w, \mathbb{D}, \Sigma) \qquad \text{where } w \in \mathit{Nat}
\end{array}
$$

$$
\begin{array}{ll}
(\sigma, w, \mathbb{D}, \Sigma) \models P & \text{iff } (\sigma, \Sigma) \models P \\
(\sigma, w, \mathbb{D}, \Sigma) \models \mathsf{arem}(\mathbb{C}') & \text{iff } \mathbb{D} = \mathbb{C}' \\
((s,h), w, \mathbb{D}, \Sigma) \models \mathsf{wf}(E) & \text{iff } \exists n.\ (\llbracket E \rrbracket_s = n) \wedge (n \leq w) \\
(\sigma, w, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{a}} & \text{iff } \exists \mathbb{D}'.\ (\sigma, w, \mathbb{D}', \Sigma) \models p \\
(\sigma, w, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{w}} & \text{iff } \exists w'.\ (\sigma, w', \mathbb{D}, \Sigma) \models p \\
(\sigma, w, \mathbb{D}, \Sigma) \models p \between q & \text{iff } \exists w_1, w_2, \mathbb{D}_1, \mathbb{D}_2.\ (\sigma, w_1, \mathbb{D}_1, \Sigma) \models p \wedge (\sigma, w_2, \mathbb{D}_2, \Sigma) \models q \\
 & \quad \wedge (w = w_1 + w_2) \wedge (\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2) \\
(\sigma, \Sigma) \models \llbracket p \rrbracket & \text{iff } \exists w, \mathbb{D}.\ (\sigma, w, \mathbb{D}, \Sigma) \models p
\end{array}
$$

$$
\mathbb{D}_1 \bot \mathbb{D}_2 \ \text{ iff } \ (\mathbb{D}_1 = \bullet) \vee (\mathbb{D}_2 = \bullet)
$$

$$
\mathbb{D}_1 \uplus \mathbb{D}_2 \stackrel{\text{def}}{=} \begin{cases} \mathbb{D}_2 & \text{if } \mathbb{D}_1 = \bullet \\ \mathbb{D}_1 & \text{if } \mathbb{D}_2 = \bullet \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

$$
(\sigma_1, w_1, \mathbb{D}_1, \Sigma_1) \uplus (\sigma_2, w_2, \mathbb{D}_2, \Sigma_2)
$$

$$
\stackrel{\text{def}}{=} \begin{cases} (\sigma_1 \uplus \sigma_2, w_1 + w_2, \mathbb{D}_1 \uplus \mathbb{D}_2, \Sigma_1 \uplus \Sigma_1) & \text{if } \sigma_1 \bot \sigma_2, \mathbb{D}_1 \bot \mathbb{D}_2 \text{ and } \Sigma_1 \bot \Sigma_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

$$
\mathcal{S} \models p * q \ \text{ iff } \ \exists \mathcal{S}_1, \mathcal{S}_2.\ (\mathcal{S} = \mathcal{S}_1 \uplus \mathcal{S}_2) \wedge (\mathcal{S}_1 \models p) \wedge (\mathcal{S}_2 \models q)
$$

$$
\begin{array}{l}
\mathsf{Sta}(p, R) \ \text{ iff} \\
\quad \forall \sigma, w, \mathbb{D}, \Sigma, \sigma', \Sigma', b.\ ((\sigma, w, \mathbb{D}, \Sigma) \models p) \wedge (((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R) \\
\quad \Longrightarrow \exists w'.\ (\sigma', w', \mathbb{D}, \Sigma') \models p \ \wedge \ (b = \mathbf{false} \implies w' = w)
\end{array}
$$

Figure 6: Semantics of assertions (part II).

2.2 Definition of RGSim-T

Definition 2 (RGSim-T).
$R, G, I \models \{P\} C \preceq \mathbb{C} \{Q\}$ iff
for all σ and Σ, if $(\sigma, \Sigma) \models P$, then there exists M such that $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$.

Whenever $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, then $(\sigma, \Sigma) \models I * \mathbf{true}$ and the following are true:

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:
   (a) either, there exist M', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$;
   (b) or, there exists M' such that $M' < M$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}, \Sigma)$;
2. for any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', M', $\mathbb{C}'$ and Σ' such that $\sigma'' = \sigma' \uplus \sigma_F$, $(\mathbb{C}, \Sigma \uplus \Sigma_F) \xrightarrow{e}{}^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$;
3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$, then there exists M' such that $R, G, I \models (C, \sigma', M') \preceq_Q (\mathbb{C}, \Sigma')$;
4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$, then $R, G, I \models (C, \sigma', M) \preceq_Q (\mathbb{C}, \Sigma')$;
5. if $C = \mathbf{skip}$, then for any Σ_F, if $\Sigma \perp \Sigma_F$, one of the following holds:
   (a) either, there exists Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbf{skip}, \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, \Sigma') \models Q$;
   (b) or, $\mathbb{C} = \mathbf{skip}$ and $(\sigma, \Sigma) \models Q$;
6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$ and $\Sigma \perp \Sigma_F$, then $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Inspired by Vafeiadis [13], we directly embed the framing aspect of separation logic in Def. 2. At each condition, we introduce the frame states σ_F and Σ_F at the target and source levels to represent the remaining parts of the states owned by other threads in the system. The commands C and $\mathbb{C}$ must not change the frame states during their executions.

Technically, we introduce theses σ_F and Σ_F quantifications to admit the frame rules (e.g., the B-FRAME rule in Fig. 7) and the parallel compositionality. Suppose we remove the frame states in Definition 2. Then consider the following example. We can prove

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models \{\mathsf{emp}\}\ ([100] := 1) \preceq ([100] := 2)\ \{\mathsf{emp}\} \tag{2.1}$$

since both programs would abort at empty states. If the frame rule holds, we would get the following by framing $[100] \mapsto 0 \wedge [100] \Rightarrow 0$ to (2.1):

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models \{[100] \mapsto 0 \wedge [100] \Rightarrow 0\}\ ([100] := 1) \preceq ([100] := 2)\ \{[100] \mapsto 0 \wedge [100] \Rightarrow 0\}$$

which obviously does not hold! (In our previous work RGSim [7], the frame rule we provided is more like an invariance rule in Hoare logic. We do not have a real frame rule due to the above reason.) Similar issue also shows up in admitting the parallel compositionality (the B-PAR rule in Fig. 7). The thread t would abort if it accesses the local state of another thread t', while the whole program may not abort with t and t' running in parallel. So we can construct a similar counterexample as (2.1) where the simulation holds for each single thread but fails for the whole program.

Here we address the above issue by embedding the framing aspect directly in the simulation definition, inspired by Vafeiadis [13]. For the simulation in Definition 2 with the σ_F and Σ_F quantifications, the above example (2.1) is no longer satisfied.

3 Logic

Inference rules are shown in Figures 7 and 8.

$$\frac{R, G, I \vdash \{P\} C_1 \preceq \mathbb{C}_1 \{P'\} \qquad R, G, I \vdash \{P'\} C_2 \preceq \mathbb{C}_2 \{Q\}}{R, G, I \vdash \{P\} C_1; C_2 \preceq \mathbb{C}_1; \mathbb{C}_2 \{Q\}} \text{ (B-SEQ)}$$

$$\frac{P \Rightarrow (B \Leftrightarrow \mathbb{B}) * I \qquad R, G, I \vdash \{P \wedge B\} C_1 \preceq \mathbb{C}_1 \{Q\} \qquad R, G, I \vdash \{P \wedge \neg B\} C_2 \preceq \mathbb{C}_2 \{Q\}}{R, G, I \vdash \{P\} \textbf{if } (B)\ C_1 \textbf{ else } C_2 \preceq \textbf{if } (\mathbb{B})\ \mathbb{C}_1 \textbf{ else } \mathbb{C}_2 \{Q\}} \text{ (B-IF)}$$

$$\frac{P \Rightarrow (B \Leftrightarrow \mathbb{B}) * I \qquad R, G, I \vdash \{P \wedge B\} C \preceq \mathbb{C} \{P\}}{R, G, I \vdash \{P\} \textbf{while } (B)\ C \preceq \textbf{while } (\mathbb{B})\ \mathbb{C} \{P \wedge \neg B\}} \text{ (B-WHILE)}$$

$$\frac{\begin{array}{c} R \vee G_2, G_1, I \vdash \{P_1 * P\} C_1 \preceq \mathbb{C}_1 \{Q_1 * Q_1'\} \qquad R \vee G_1, G_2, I \vdash \{P_2 * P\} C_2 \preceq \mathbb{C}_2 \{Q_2 * Q_2'\} \\ P \vee Q_1' \vee Q_2' \Rightarrow I \qquad I \triangleright R \end{array}}{R, G_1 \vee G_2, I \vdash \{P_1 * P_2 * P\} C_1 \,\|\, C_2 \preceq \mathbb{C}_1 \,\|\, \mathbb{C}_2 \{Q_1 * Q_2 * (Q_1' \wedge Q_2')\}} \text{ (B-PAR)}$$

$$\frac{}{\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{P\} \textbf{skip} \preceq \textbf{skip} \{P\}} \text{ (B-SKIP)} \qquad \frac{P \Rightarrow (E = \mathbb{E})}{\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \vdash \{P\} \textbf{print}(E) \preceq \textbf{print}(\mathbb{E}) \{P\}} \text{ (B-PRT)}$$

$$\frac{R, G, I \vdash \{P\} C \preceq \mathbb{C} \{Q\} \qquad G^+ \Rightarrow G \qquad \mathsf{Sta}(P', (R')^+ * \mathsf{Id}) \qquad I' \triangleright \{R', G'\} \qquad P' \Rightarrow I' * \textbf{true}}{R * R', G * G', I * I' \vdash \{P * P'\} C \preceq \mathbb{C} \{Q * P'\}} \text{ (B-FRAME)}$$

$$\frac{\begin{array}{c} R_1, G_1, I_1 \vdash \{P_1\} C \preceq \mathrm{C_M} \{Q_1\} \qquad R_2, G_2, I_2 \vdash \{P_2\} \mathrm{C_M} \preceq \mathbb{C} \{Q_2\} \\ \mathsf{MPrecise}(I_1, I_2) \qquad ((G_1)^+ \mathbin{\hat{\S}} (G_2)^+) \ \Rightarrow\ (G_1 \mathbin{\hat{\S}} G_2)^+ \qquad (R_1 \mathbin{\tilde{\S}} R_2)^+ \ \Rightarrow\ ((R_1)^+ \mathbin{\tilde{\S}} (R_2)^+) \end{array}}{(R_1 \mathbin{\tilde{\S}} R_2), (G_1 \mathbin{\hat{\S}} G_2), (I_1 \mathbin{\S} I_2) \vdash \{P_1 \mathbin{\S} P_2\} C \preceq \mathbb{C} \{Q_1 \mathbin{\S} Q_2\}} \text{ (TRANS)}$$

$$\frac{R, G, I \vdash \{P \wedge \mathsf{arem}(\mathbb{C})\} C \{Q \wedge \mathsf{arem}(\textbf{skip})\}}{R, G, I \vdash \{P\} C \preceq \mathbb{C} \{Q\}} \text{ (U2B)}$$

Figure 7: Selected binary inference rules.

Definition 3 (Abstract Step "Implication").
$p \overset{G}{\Rrightarrow}^+ q$ iff,
for any σ, w, $\mathbb{D}$, Σ and Σ_F, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$ and $\Sigma \perp \Sigma_F$, then
there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$,
$((\sigma, \Sigma), (\sigma, \Sigma'), \textbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{C}', \Sigma') \models q$.

We also define the following syntactic sugars:

$$p \Rrightarrow^+ q \ \text{ iff } \ p \overset{\mathsf{Emp}}{\Longrightarrow}^+ q \qquad p \overset{G}{\Rrightarrow}^0 q \ \text{ iff } \ p \Rightarrow q \qquad p \Rrightarrow^0 q \ \text{ iff } \ p \Rightarrow q$$
$$p \overset{G}{\Rrightarrow}^* q \ \text{ iff } \ p \overset{G}{\Rrightarrow}^+ q \vee p \overset{G}{\Rrightarrow}^0 q \qquad p \Rrightarrow^* q \ \text{ iff } \ p \Rrightarrow^+ q \vee p \Rrightarrow^0 q$$

Note that here we introduce the Σ_F quantification similar to Definition 2 for RGSim-T. In our CSL-LICS'14 paper, we simplified the above definition and only defined $p \Rrightarrow^+ q$ to save space. The more general case $p \overset{G}{\Rrightarrow}^+ q$ defined here is useful in the A-CONSEQ rule, which is omitted in our CSL-LICS'14 paper.

We prove a few properties of $p \overset{G}{\Rrightarrow}^+ q$, as shown in Figure 9. For instance, the first rule says, we can derive $(P \wedge \mathsf{arem}(\mathbb{C})) \Rrightarrow^+ (Q \wedge \mathsf{arem}(\textbf{skip}) \wedge \mathsf{wf}(E))$ by executing the source code $\mathbb{C}$. And since the source

$$\frac{}{\mathsf{Emp},\mathsf{Emp},\mathsf{emp} \vdash \{p\}\mathbf{skip}\{p\}}\ (\textsc{skip}) \qquad \frac{\vdash_{\text{SL}} [p]c[q] \qquad c \text{ is silent}}{\mathsf{Emp},\mathsf{Emp},\mathsf{emp} \vdash \{p\}c\{q\}}\ (\textsc{env})$$

$$\frac{\vdash_{\text{SL}} [p]C[q] \qquad (\!|p|\!| \ltimes \!|q|\!|) \Rightarrow G * \mathsf{True} \qquad I \triangleright G \qquad p \vee q \Rightarrow I * \mathbf{true}}{[I], G, I \vdash \{p\}\langle C\rangle\{q\}}\ (\textsc{atom})$$

$$\frac{\begin{array}{c} p \Rrightarrow^{a} p' \qquad \vdash_{\text{SL}} [p']C[q'] \qquad q' \Rrightarrow^{b} q \qquad + \in \{a, b\} \\ (\!|p|\!| \propto \!|q|\!|) \Rightarrow G * \mathsf{True} \qquad I \triangleright G \qquad p \vee q \Rightarrow I * \mathbf{true} \end{array}}{[I], G, I \vdash \{p\}\langle C\rangle\{q\}}\ (\textsc{atom}^{+})$$

$$\frac{[I], G, I \vdash \{p\}\langle C\rangle\{q\} \qquad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \qquad I \triangleright R}{R, G, I \vdash \{p\}\langle C\rangle\{q\}}\ (\textsc{atom-r})$$

$$\frac{R, G, I \vdash \{p\}C_1\{p'\} \qquad R, G, I \vdash \{p'\}C_2\{q\}}{R, G, I \vdash \{p\}C_1; C_2\{q\}}\ (\textsc{seq})$$

$$\frac{p \Rightarrow (B = B) * I \qquad p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp}) \qquad R, G, I \vdash \{p'\}C\{p\}}{R, G, I \vdash \{p\}\mathbf{while}\ (B)\ C\{p \wedge \neg B\}}\ (\textsc{while})$$

$$\frac{R, G, I \vdash \{p\}C\{q\}}{R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{w}}\}C\{\lfloor q \rfloor_{\mathsf{w}}\}}\ (\textsc{hide-w})$$

$$\frac{R, G, I \vdash \{p\}C\{q\} \qquad \mathsf{Sta}(p', (R')^{+} * \mathsf{Id}) \qquad I' \triangleright \{R', G'\} \qquad p' \Rightarrow I' * \mathbf{true} \qquad G^{+} \Rightarrow G}{R * R', G * G', I * I' \vdash \{p * p'\}C\{q * p'\}}\ (\textsc{frame})$$

$$\frac{R, G, I \vdash \{p\}C\{q\} \qquad \mathsf{Sta}(p', \{R^{+} * \mathsf{Id}, G * \mathsf{True}\})}{R, G, I \vdash \{p \circledcirc p'\}C\{q \circledcirc p'\}}\ (\textsc{fr-conj})$$

$$\frac{R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_1)\}C\{\lfloor q \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_2)\}}{R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_1; \mathbb{C}_3)\}C\{\lfloor q \rfloor_{\mathsf{a}} \wedge \mathsf{arem}(\mathbb{C}_2; \mathbb{C}_3)\}}\ (\textsc{arem})$$

$$\frac{R, G, I \vdash \{p_1\}C\{q_1\} \qquad R, G, I \vdash \{p_2\}C\{q_2\}}{R, G, I \vdash \{p_1 \vee p_2\}C\{q_1 \vee q_2\}}\ (\textsc{disj})$$

$$\frac{p \overset{G}{\Rightarrow}{}^{*} p' \qquad R, G, I \vdash \{p'\}C\{q'\} \qquad q' \overset{G}{\Rightarrow}{}^{*} q \qquad \mathsf{Sta}(\{p, q\}, R * \mathsf{Id}) \qquad p \vee q \Rightarrow I * \mathbf{true}}{R, G, I \vdash \{p\}C\{q\}}\ (\textsc{a-conseq})$$

Figure 8: Selected unary inference rules.

code makes multiple steps, we are allowed to increase the number of tokens ($\mathsf{wf}(E)$). We can also execute the source code in trivial cases, for example, when the source code is $\mathbf{skip}; \mathbb{C}$, or it is a while loop but we know for sure the value of the loop condition. In those cases, the step of the source code is an identity transition. Moreover, $p \overset{G}{\Rightarrow}^+ q$ is transitive and we can also have "frame rule" (i.e., local reasoning) over it.

$$\frac{\mathbb{C} \neq \mathbf{skip} \qquad \vdash_{\text{SL}} [P]\mathbb{C}[Q]}{(P \wedge \mathsf{arem}(\mathbb{C})) \Rrightarrow^+ (Q \wedge \mathsf{arem}(\mathbf{skip}) \wedge \mathsf{wf}(E))}$$

$$\frac{P \Rightarrow I * \mathbf{true}}{(P \wedge \mathsf{arem}(\mathbf{skip}; \mathbb{C})) \overset{[I]}{\Longrightarrow}^+ (P \wedge \mathsf{arem}(\mathbb{C}) \wedge \mathsf{wf}(E))}$$

$$\frac{P \Rightarrow \mathbb{B} * I}{(P \wedge \mathsf{arem}(\mathbf{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathbf{else}\ \mathbb{C}_2)) \overset{[I]}{\Longrightarrow}^+ (P \wedge \mathsf{arem}(\mathbb{C}_1) \wedge \mathsf{wf}(E))}$$

$$\frac{P \Rightarrow (\neg\mathbb{B}) * I}{(P \wedge \mathsf{arem}(\mathbf{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathbf{else}\ \mathbb{C}_2)) \overset{[I]}{\Longrightarrow}^+ (P \wedge \mathsf{arem}(\mathbb{C}_2) \wedge \mathsf{wf}(E))}$$

$$\frac{P \Rightarrow \mathbb{B} * I}{(P \wedge \mathsf{arem}(\mathbf{while}\ (\mathbb{B})\ \mathbb{C})) \overset{[I]}{\Longrightarrow}^+ (P \wedge \mathsf{arem}(\mathbb{C}; \mathbf{while}\ (\mathbb{B})\ \mathbb{C}) \wedge \mathsf{wf}(E))}$$

$$\frac{P \Rightarrow (\neg\mathbb{B}) * I}{(P \wedge \mathsf{arem}(\mathbf{while}\ (\mathbb{B})\ \mathbb{C})) \overset{[I]}{\Longrightarrow}^+ (P \wedge \mathsf{arem}(\mathbf{skip}) \wedge \mathsf{wf}(E))}$$

$$\frac{(P \wedge \mathsf{arem}(\mathbb{C}_1)) \overset{G}{\Rightarrow}^+ (Q \wedge \mathsf{arem}(\mathbb{C}_2) \wedge \mathsf{wf}(E))}{(P \wedge \mathsf{arem}(\mathbb{C}_1; \mathbb{C}_3)) \overset{G}{\Rightarrow}^+ (Q \wedge \mathsf{arem}(\mathbb{C}_2; \mathbb{C}_3) \wedge \mathsf{wf}(E))}$$

$$\frac{p \overset{G}{\Rightarrow}^+ p' \qquad p' \overset{G}{\Rightarrow}^+ q \qquad I \triangleright G}{p \overset{G}{\Rightarrow}^+ q} \qquad \frac{p \Rightarrow p' \qquad p' \overset{G'}{\Rightarrow}^+ q' \qquad q' \Rightarrow q \qquad G' \Rightarrow G}{p \overset{G}{\Rightarrow}^+ q}$$

$$\frac{p_1 \overset{G}{\Rightarrow}^+ q_1 \qquad p_2 \overset{G}{\Rightarrow}^+ q_2}{(p_1 \vee p_2) \overset{G}{\Rightarrow}^+ (q_1 \vee q_2)} \qquad \frac{p \overset{G}{\Rightarrow}^+ q}{(p * p') \overset{G}{\Rightarrow}^+ (q * p')}$$

Figure 9: Properties of $p \overset{G}{\Rightarrow} q$.

Below we discuss some interesting rules which are not shown in our CSL-LICS'14 paper due to the space limit. The binary rules are very similar to those in our previous work RGSim [7]. The TRANS rule shows the transitivity of our RGSim-T relation.

For the unary rules in Figure 8, in addition to rules for atomic blocks, we have SKIP and ENV rules to reason about **skip** and primitive instructions. Here we assume the unary logic handles only programs which do not produce external events (e.g., the ENV rule has a side condition saying that "c is silent"). For commands producing events, such as the print command, we require *lockstep* at the target and source levels and prove such refinement using the binary inference rules (e.g., the B-PRT rule in Figure 7). It is also possible to extend the current unary logic with assertions for event traces and provide unary rules to reason about commands with events. Note that although the shared resource is empty in the SKIP and ENV rules, we can derive rules allowing resource sharing from them and the FRAME rule in Figure 8.

In addition to the rules for while loops as in the CSL-LICS'14 paper, we also have unary rules for sequential composition (the SEQ rule in Figure 8) and for if-then-else composition (omitted here), both of which are in the same forms as in LRG [2]. The unary FRAME rule is similar to the binary one in Figure 7. It is also in the same form as in LRG [2].

The FR-CONJ rule is like the frame rule in RGSep [12]. The frame p' may specify the number of tokens used by the context of the code C, i.e., the code C does not consume these tokens in p'. The frame p' may also specify the shared concrete and abstract states (and the case usually occurs when the number of tokens depends on the concrete and abstract states). So we use the new operator $\textcircled{0}$ to ensure that the concrete and abstract states specified in p and p' coincide.

The AREM rule is like a frame rule over source code. It allows us to reason about refinement using "local" source code, i.e., source code which is really refined by the target.

The A-CONSEQ rule allows us to execute the source code outside of an atomic block. It requires that the transitions of the source code over the shared states satisfy G^+, but it is usually used when the steps are simply identity transitions. For instance, we can use the rule to unfold a while loop at the source at any time in a refinement proof (we do not have to be in an atomic block of the target code). When $p \overset{G}{\Rightarrow}^* p'$ and $q' \overset{G}{\Rightarrow}^* q$ are $p \Rightarrow p'$ and $q' \Rightarrow q$ respectively, this rule becomes the normal CONSEQ rule (see RGSep [12] and LRG [2]).

We can also *derive* the following WHILE-TERM rule from the WHILE rule. The derivation is shown in Section 5.

$$\frac{\begin{array}{c} R, G, I \vdash \{p \wedge B \wedge (E = \alpha)\} C \{p \wedge (E < \alpha)\} \qquad p \wedge B \Rightarrow E > 0 \\ p \Rightarrow ((B = B) \wedge (E = E)) * I \qquad G^+ \Rightarrow G \qquad \alpha \text{ is a fresh logical variable} \end{array}}{R, G, I \vdash \{\lfloor p \rfloor_{\mathtt{w}}\} \textbf{while } (B)\ C \{\lfloor p \rfloor_{\mathtt{w}} \wedge \neg B\}} \text{ (WHILE-TERM)}$$

The WHILE-TERM rule is similar to a total correctness while rule (e.g., see [10]). In every round of the loop, the loop variant E decreases (but should always be positive). We can verify refinement for such a locally-terminating loop (a loop that always terminates regardless of environment steps) without specifying tokens. To derive this rule, we actually need to introduce the number of tokens as an auxiliary state for the loop iterations and relate it to the loop variant E in the real state.

Soundness of the logic is proved in Section 5 (where we also define the unary judgment semantics).

4 Examples

In this section, we verify the examples claimed in our CSL-LICS'14 paper (see Figure 10). To simplify the presentation of the proofs, assume we always have the ownerships of program variables.

Linearizability & Lock-Freedom	Counter and its variants
	Treiber stack
	Michael-Scott lock-free queue [8]
	DGLM lock-free queue [1]
Non-Atomic Object Correctness	Synchronous queue [9]
Correctness of Optimized Algo (Equivalence)	Counter vs. its variants
	TAS lock vs. TTAS lock [3]

Figure 10: Verified examples using our logic.

4.1 Counter and Its Variants

In Figure 11, we show four possible implementations of the counter. Though they are quite simple, they illustrate different choices that programmers may make to implement a concurrent object. The abstract atomic `INC` operation is shown below:

```
INC() {   X := X + 1;  }
```

```
1 inc() {
2  local t, b;
3  b := false;
4  while (!b) {
5    < t := x; >
6    b := cas(&x, t, t+1);
7  }
8 }
```

```
1 inc'() {
2  local t, b;
3  b := false;
4  < t := x; >
5  while (!b) {
6    b := cas(&x, t, t+1);
7    < t := x; >
8  }
9 }
```

```
 1 incOpt() {
 2  local t, b, b';
 3  b := false;
 4  while (!b) {
 5    b' := false;
 6    while (!b') {
 7      < t := x; >
 8      < b' := (t = x); >
 9    }
10    b := cas(&x, t, t+1);
11  }
12 }
```

```
 1 incOpt'() {
 2  local t, b, b';
 3  b := false;
 4  while (!b) {
 5    < t := x; >
 6    < b' := (t = x); >
 7    while (!b') {
 8      < t := x; >
 9      < b' := (t = x); >
10    }
11    b := cas(&x, t, t+1);
12  }
13 }
```

Figure 11: Various implementations of counter.

Below we first verify that each implementation C of the counter is correct w.r.t. to `INC`. Here correctness refer to linearizability and lock-freedom together. As explained in the submitted paper, we only need to prove the following in our logic:

$$R, G, I \vdash \{I\}\ C \preceq \texttt{INC}\ \{I\}$$

where R and G specify the possible actions (i.e., increments) on the well-formed shared data structure (i.e., counter) fenced by I. In all these examples, they share the same R, G and I as follows:

$$I \stackrel{\text{def}}{=} (\mathtt{x} = \mathtt{X}) \qquad R = G \stackrel{\text{def}}{=} (I \propto I) \vee [I]$$

By the U2B rule, the above is reduced to proving the following unary judgment:

$$R, G, I \vdash \{I \wedge \mathsf{arem}(\mathtt{X} := \mathtt{X} + 1)\} C \{I \wedge \mathsf{arem}(\mathbf{skip})\}$$

The proofs are shown in Figures 12, 13, 14 and 15.

We can also prove the equivalence between `incOpt` and `inc`. That is, we prove:

$$R, G, I \vdash \{I\}\ \mathtt{incOpt} \preceq \mathtt{inc}\ \{I\} \qquad \text{and} \qquad R, G, I \vdash \{I\}\ \mathtt{inc} \preceq \mathtt{incOpt}\ \{I\}$$

Here we use the same R, G and I as above (always use `x` at the left side and `X` at the right side). The proofs are shown in Figures 17 and 18. The equivalence between `incOpt'` and `inc` is similar.

```
1 inc() {
2  local t, b;
   {I ∧ arem(X := X + 1)}
3  b := false;
   {(¬b ∧ I ∧ arem(X := X + 1)) ∨ (b ∧ I ∧ arem(skip))}     //Applying the WHILE rule and the HIDE-W rule
4  while (!b) {
     {¬b ∧ I ∧ arem(X := X + 1) ∧ wf(0)}
     {x = X}  ∗ (emp ∧ ¬b ∧ arem(X := X + 1) ∧ wf(0))       //Applying the FRAME rule
5    < t := x; >
     {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(1))}  ∗ (emp ∧ ¬b ∧ arem(X := X + 1) ∧ wf(0))
     {(¬b ∧ (x = X = t) ∧ arem(X := X + 1) ∧ wf(0))
      ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1) ∧ wf(1))}
6    b := cas(&x, t, t+1);
     {(b ∧ I ∧ arem(skip) ∧ wf(1)) ∨ (¬b ∧ I ∧ arem(X := X + 1) ∧ wf(1))}
7  }
   {I ∧ arem(skip)}
8 }
```

Figure 12: Proving `inc` refines `INC`.

```
1 inc'() {
2  local t, b;
   {I ∧ arem(X := X + 1)}
3  b := false;
   {¬b ∧ I ∧ arem(X := X + 1)}
4  < t := x; >
   {(¬b ∧ (x = X = t) ∧ arem(X := X + 1))
    ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1))
    ∨ (b ∧ I ∧ arem(skip))}     //Applying the WHILE rule and the HIDE-W rule
5  while (!b) {
     {(¬b ∧ (x = X = t) ∧ arem(X := X + 1) ∧ wf(0)) ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1) ∧ wf(1))}
6    b := cas(&x, t, t+1);
     {(b ∧ I ∧ arem(skip) ∧ wf(1)) ∨ (¬b ∧ I ∧ arem(X := X + 1) ∧ wf(1))}
7    < t := x; >
     {(¬b ∧ (x = X = t) ∧ arem(X := X + 1) ∧ wf(1))
      ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1) ∧ wf(2))
      ∨ (b ∧ I ∧ arem(skip) ∧ wf(1))}
8  }
   {I ∧ arem(skip)}
9 }
```

Figure 13: Proving `inc'` refines `INC`.

```
1  incOpt() {
2   local t, b, b';
    {I ∧ arem(X := X + 1)}
3   b := false;
    {(¬b ∧ I ∧ arem(X := X + 1)) ∨ (b ∧ I ∧ arem(skip))}      //Applying the WHILE rule and the HIDE-W rule
4   while (!b) {
      {¬b ∧ I ∧ arem(X := X + 1) ∧ wf(1)}
      {(x = X) ∧ wf(1)} ∗ (emp ∧ ¬b ∧ arem(X := X + 1))        //Applying the FRAME rule
5     b' := false;
      {(¬b' ∧ (x = X) ∧ wf(1)) ∨ (b' ∧ (x = X = t)) ∨ (b' ∧ (x = X ≠ t) ∧ wf(2))}      //Applying the WHILE rule
6     while (!b') {
        {(x = X) ∧ wf(0)}
7       < t := x; >
        {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(1))}
8       < b' := (t = x); >
        {(b' ∧ (x = X = t)) ∨ (b' ∧ (x = X ≠ t) ∧ wf(2)) ∨ (¬b' ∧ (x = X ≠ t) ∧ wf(1))}
9     }
      {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(2))} ∗ (emp ∧ ¬b ∧ arem(X := X + 1))
      {(¬b ∧ (x = X = t) ∧ arem(X := X + 1) ∧ wf(0))
       ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1) ∧ wf(2))}
10    b := cas(&x, t, t+1);
      {(b ∧ I ∧ arem(skip) ∧ wf(1)) ∨ (¬b ∧ I ∧ arem(X := X + 1) ∧ wf(2))}
11  }
    {I ∧ arem(skip)}
12 }
```

Figure 14: Proving `incOpt` refines `INC`.

```
1  incOpt'() {
2   local t, b, b';
    {I ∧ arem(X := X + 1)}
3   b := false;
    {(¬b ∧ I ∧ arem(X := X + 1)) ∨ (b ∧ I ∧ arem(skip))}      //Applying the WHILE rule and the HIDE-W rule
4   while (!b) {
      {¬b ∧ I ∧ arem(X := X + 1) ∧ wf(0)}
      {x = X} ∗ (emp ∧ ¬b ∧ arem(X := X + 1) ∧ wf(0))          //Applying the FRAME rule
5     < t := x; >
      {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(1))}
6     < b' := (t = x); >
      {(b' ∧ (x = X = t)) ∨ ((x = X ≠ t) ∧ wf(1))}      //Applying the WHILE rule
7     while (!b') {
        {(x = X) ∧ wf(0)}
8       < t := x; >
        {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(1))}
9       < b' := (t = x); >
        {(b' ∧ (x = X = t)) ∨ ((x = X ≠ t) ∧ wf(1))}
10    }
      {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(1))} ∗ (emp ∧ ¬b ∧ arem(X := X + 1) ∧ wf(0))
      {(¬b ∧ (x = X = t) ∧ arem(X := X + 1) ∧ wf(0))
       ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1) ∧ wf(1))}
11    b := cas(&x, t, t+1);
      {(b ∧ I ∧ arem(skip) ∧ wf(1)) ∨ (¬b ∧ I ∧ arem(X := X + 1) ∧ wf(1))}
12  }
    {I ∧ arem(skip)}
13 }
```

Figure 15: Proving `incOpt'` refines `INC`.

$I \stackrel{\text{def}}{=} (\mathtt{x} = \mathtt{X})$
$R = G \stackrel{\text{def}}{=} (\exists n.\ (\mathtt{x} = \mathtt{X} = n) \propto (\mathtt{x} = \mathtt{X} > n)) \vee [I]$

```
1  incOpt'() {
2   local t, b;
    {I ∧ arem(X := X + 1)}
3   b := false;
    {(¬b ∧ I ∧ arem(X := X + 1)) ∨ (b ∧ I ∧ arem(skip))}      //Applying the WHILE rule and the HIDE-W rule
4   while (!b) {
      {¬b ∧ I ∧ arem(X := X + 1) ∧ wf(0)}
      {x = X}  ∗ (emp ∧ ¬b ∧ arem(X := X + 1) ∧ wf(0))        //Applying the FRAME rule
5     < t := x; >
      {(x = X = t = α) ∨ ((x = X > α) ∧ (t = α) ∧ wf(1))}
6     < b' := (t = x); >
      {(b' ∧ (x = X = t = α)) ∨ ((x = X > α) ∧ (t = α) ∧ wf(1))}
      {(b' ∧ (x = X = t = α)) ∨ (x = X > α)}   ⍉((x = X = α) ∨ (x = X > α) ∧ wf(1))
            //Applying the FR-CONJ rule      //Applying the WHILE rule and the HIDE-W rule
7     while (!b') {
        {(x = X > α) ∧ wf(0)}
8       < t := x; >
        {(x = X = t > α) ∨ ((x = X > t > α) ∧ wf(1))}
9       < b' := (t = x); >
        {(b' ∧ (x = X = t > α)) ∨ ((x = X > t > α) ∧ wf(1))}
        {(b' ∧ (x = X = t ≥ α)) ∨ ((x = X > α) ∧ wf(1))}
10    }
      {(x = X = t = α) ∨ (x = X > α)}   ⍉((x = X = α) ∨ (x = X > α) ∧ wf(1))
      {(x = X = t = α) ∨ ((x = X > α) ∧ wf(1))}
      {(x = X = t) ∨ ((x = X ≠ t) ∧ wf(1))}   ∗ (emp ∧ ¬b ∧ arem(X := X + 1) ∧ wf(0))
      { (¬b ∧ (x = X = t) ∧ arem(X := X + 1) ∧ wf(0))      }
      {  ∨ (¬b ∧ (x = X ≠ t) ∧ arem(X := X + 1) ∧ wf(1)) }
11    b := cas(&x, t, t+1);
      {(b ∧ I ∧ arem(skip) ∧ wf(1)) ∨ (¬b ∧ I ∧ arem(X := X + 1) ∧ wf(1))}
12  }
    {I ∧ arem(skip)}
13 }
```

Figure 16: Proving `incOpt'` refines `INC` (an alternative approach by using the FR-CONJ rule). α is a logical variable.

$\texttt{inc} \stackrel{\text{def}}{=} (\texttt{B := false; incLoop;})$
$\texttt{incLoop} \stackrel{\text{def}}{=} (\texttt{while(!B) \{ <T:=X>; incCas; \}})$
$\texttt{incCas} \stackrel{\text{def}}{=} (\texttt{B := cas(\&X, T, T+1);})$

```
1  incOpt() {
2    local t, b, b';
     {I ∧ arem(inc)}
3    b := false;
     {(¬b ∧ ¬B ∧ I ∧ arem(incLoop)) ∨ (b ∧ B ∧ I ∧ arem(skip))}
                //Applying the WHILE rule and the HIDE-W rule
4    while (!b) {
       {¬b ∧ ¬B ∧ I ∧ arem(incLoop) ∧ wf(0)}
5      b' := false;
       {(¬b' ∧ ¬b ∧ ¬B ∧ (x = X) ∧ arem(incLoop) ∧ wf(0))
        ∨ (b' ∧ ¬b ∧ ¬B ∧ (x = X) ∧ (t = T) ∧ arem(incCas;incLoop) ∧ wf(0))}
                  //Applying the WHILE rule and the HIDE-W rule
6      while (!b') {
         {¬b' ∧ ¬b ∧ ¬B ∧ (x = X) ∧ arem(incLoop) ∧ wf(0)}
         {¬b ∧ ¬B ∧ (x = X) ∧ arem(<T:=X>; incCas; incLoop) ∧ wf(1)}
7        < t := x; >
         {¬b ∧ ¬B ∧ (x = X) ∧ (t = T) ∧ arem(incCas; incLoop) ∧ wf(1)}
8        < b' := (t = x); >
         {(¬b' ∧ ¬b ∧ ¬B ∧ (x = X) ∧ arem(incLoop) ∧ wf(1))
          ∨ (b' ∧ ¬b ∧ ¬B ∧ (x = X) ∧ (t = T) ∧ arem(incCas;incLoop) ∧ wf(1))}
9      }
       {b' ∧ (x = X) ∧ (t = T) ∧ arem(incCas;incLoop) ∧ wf(0)}
10     b := cas(&x, t, t+1);
       {(b = B) ∧ I ∧ arem(incLoop) ∧ wf(1)}
       {(b ∧ B ∧ I ∧ arem(skip)) ∨ (¬b ∧ ¬B ∧ I ∧ arem(incLoop) ∧ wf(1))}
11   }
     {I ∧ arem(skip)}
12 }
```

Figure 17: Proving `incOpt` refines `inc`.

```
incOpt ≝ (B := false; incOptLoop;)
incOptLoop ≝ (while(!B) { incOptInner; incCas; })
incOptInner ≝ (B':=false; while(!B') { <T:=X>; <B':=(T=X)>; })
incCas ≝ (B := cas(&X, T, T+1);)

1 inc() {
2  local t, b;
   {I ∧ arem(incOpt)}
3  b := false;
   {(¬b ∧ ¬B ∧ I ∧ arem(incOptLoop)) ∨ (b ∧ B ∧ I ∧ arem(skip))}
            //Applying the WHILE rule and the HIDE-W rule
4  while (!b) {
     {¬b ∧ ¬B ∧ I ∧ arem(incOptLoop) ∧ wf(0)}
5    < t := x; >
     {¬b ∧ ¬B ∧ (x = X) ∧ (t = T) ∧ arem(incCas; incOptLoop) ∧ wf(1)}
6    b := cas(&x, t, t+1);
     {(b = B) ∧ I ∧ arem(incOptLoop) ∧ wf(1)}
7  }
   {I ∧ arem(skip)}
8 }
```

Figure 18: Proving `inc` refines `incOpt`.

4.2 TAS Lock and TTAS Lock

```
1 lock() {
2  local b, b';
3  b := true;
4  while (b) {
5    < b' := l; >
6    while (b') {
7      < b' := l; >
8    }
9    b := getAndSet(&l, true);
10  }
11 }

1 unlock() {
2  < l := false; >
3 }
```

```
1 LOCK() {
2  local B;
3  B := getAndSet(&L, true);
4  while (B) {
5    B := getAndSet(&L, true);
6  }
7 }

1 UNLOCK() {
2  < L := false; >
3 }
```

Figure 19: `TTASLock` (the left) and `TASLock` (the right).

In Figure 19, we show the implementations of TTAS lock and TAS lock [3]. We can prove the equivalence between these two implementations. That is, we prove:

$$R, G, I \vdash \{I\}\ \texttt{lock} \preceq \texttt{LOCK}\ \{I\} \quad \text{and} \quad R, G, I \vdash \{I\}\ \texttt{LOCK} \preceq \texttt{lock}\ \{I\}$$
$$R, G, I \vdash \{I\}\ \texttt{unlock} \preceq \texttt{UNLOCK}\ \{I\} \quad \text{and} \quad R, G, I \vdash \{I\}\ \texttt{UNLOCK} \preceq \texttt{unlock}\ \{I\}$$

As in the example of counters, R and G specify the possible actions on the well-formed shared data structure fenced by I. Here R, G and I can be defined as follows:

$$I \stackrel{\text{def}}{=} (\texttt{l} = \texttt{L}) \qquad R = G \stackrel{\text{def}}{=} (I \propto I) \vee [I]$$

The proofs for the refinements between `unlock` and `UNLOCK` are straightforward since their code is the same. We show the proofs for the refinements between `lock` and `LOCK` in Figures 20 and 21.

GAS $\stackrel{\text{def}}{=}$ (B := getAndSet(&L, true))
LoopGAS $\stackrel{\text{def}}{=}$ (while(B) GAS;)

```
1  lock() {
2   local b, b';
    {I ∧ arem(LOCK)}
3   b := true;
    {(b ∧ I ∧ arem(GAS; LoopGAS)) ∨ (¬b ∧ I ∧ arem(skip))}      //Applying the WHILE rule and the HIDE-W rule
4   while (b) {
      {b ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(0)}
5     < b' := l; >
      {(b ∧ b' ∧ B ∧ I ∧ arem(LoopGAS) ∧ wf(1)) ∨ (b ∧ ¬b' ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(0))}
      {b ∧ I ∧ arem(GAS; LoopGAS)}      //Applying the WHILE rule and the HIDE-W rule
6     while (b') {
        {b ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(0)}
7       < b' := l; >
        {(b ∧ b' ∧ B ∧ I ∧ arem(LoopGAS) ∧ wf(1)) ∨ (b ∧ ¬b' ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(0))}
        {(b ∧ b' ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(1)) ∨ (b ∧ ¬b' ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(0))}
8     }
      {b ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(0)}
9     b := getAndSet(&l, true);
      {(b = B) ∧ I ∧ arem(LoopGAS) ∧ wf(1)}
      {(¬b ∧ I ∧ arem(skip) ∧ wf(1)) ∨ (b ∧ I ∧ arem(GAS; LoopGAS) ∧ wf(1))}
10  }
    {I ∧ arem(skip)}
11 }
```

Figure 20: Proving TTASLock refines TASLock.

loopTTAS $\stackrel{\text{def}}{=}$ (while(b) {...})

```
1  LOCK() {
2   local B;
    {I ∧ arem(lock)}
3   B := getAndSet(&L, true);
    {(b = B) ∧ I ∧ arem(loopTTAS) ∧ wf(1)}
    {(b = B) ∧ I ∧ arem(loopTTAS)}      //Applying the WHILE rule and the HIDE-W rule
4   while (B) {
      {b ∧ B ∧ I ∧ arem(loopTTAS) ∧ wf(0)}
5     B := getAndSet(&L, true);
      {(b = B) ∧ I ∧ arem(loopTTAS) ∧ wf(1)}
6   }
    {¬b ∧ ¬B ∧ I ∧ arem(loopTTAS)}
    {I ∧ arem(skip)}
7  }
```

Figure 21: Proving TASLock refines TTASLock.

4.3 Treiber Stack

```
 1 push(v) {
 2  local x, t, b;
 3  b := false;
 4  x := cons(v, null);
 5  while (!b) {
 6    < t := S; >
 7    x.next := t;
 8    b := cas(&S, t, x);
 9  }
10 }
```

```
 1 pop() {
 2  local v, x, t, b;
 3  b := false;
 4  while (!b) {
 5    < t := S; >
 6    if (t = null) {
 7      v := EMPTY;
 8      b := true;
 9    } else {
10      v := t.data;
11      x := t.next;
12      b := cas(&S, t, x);
13    }
14  }
15  return v;
16 }
```

```
 1 PUSH(V) {
 2  < Stk := V :: Stk; >
 3 }
```

```
 1 POP() {
 2  local V;
 3  < if (Stk = ϵ) {
 4      V := EMPTY;
 5    } else {
 6      V := head(Stk);
 7      Stk := tail(Stk);
 8    }
 9  >
10  return V;
11 }
```

Figure 22: Treiber stack.

In Figure 22, we show the implementation of Treiber stack (at the left of the figure), and the abstract atomic operations (at the right). The abstract `PUSH` and `POP` operations manipulate an abstract mathematical list `Stk`, and when popping from an empty stack, `POP` returns `EMPTY`.

Below we use our logic to prove the linearizability and lock-freedom together of Treiber stack. As explained in the submitted paper, we only need to prove the following in our logic:

$$R, G, I \vdash \{I \wedge (\mathtt{v} = \mathtt{V})\}\ \mathtt{push(v)} \preceq \mathtt{PUSH(V)}\ \{I\} \qquad \text{and} \qquad R, G, I \vdash \{I\}\ \mathtt{pop} \preceq \mathtt{POP}\ \{I \wedge (\mathtt{v} = \mathtt{V})\}$$

By the U2B rule, the above is reduced to proving the following unary judgment:

$$R, G, I \vdash \{I \wedge \mathsf{arem}(\mathtt{PUSH(V)}) \wedge (\mathtt{v} = \mathtt{V})\}\ \mathtt{push(v)}\ \{I \wedge \mathsf{arem}(\boldsymbol{skip})\}$$
$$\text{and} \qquad R, G, I \vdash \{I \wedge \mathsf{arem}(\mathtt{POP})\}\ \mathtt{pop}\ \{I \wedge \mathsf{arem}(\boldsymbol{skip}) \wedge (\mathtt{v} = \mathtt{V})\}$$

We define the precise invariant I, the rely R and the guarantee G in Figure 23. The invariant I in Figure 23 maps the value sequence A of the concrete list pointed to by `S` (denoted by $(\mathtt{S} = x) * \mathsf{ls}(x, A, \mathtt{null})$) to the abstract stack `Stk`. To ensure there is no "ABA" problem [3], we follow Turon and Wand [11] and introduce a write-only auxiliary variable `GN` to remember the nodes which used to be on the stack but no longer are. The precise invariant for shared states should include those garbage nodes (garb). `GN` does not affect the behaviors of the implementation and is introduced for verification only.

$$I \stackrel{\text{def}}{=} \exists x, A.\ (\mathtt{Stk} = A) \wedge (\mathtt{S} = x) * \mathsf{ls}(x, A, \mathtt{null}) * \mathsf{garb}$$
$$\mathsf{node}(x, v, y) \stackrel{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x) \stackrel{\text{def}}{=} \mathsf{node}(x, \_, \_)$$
$$\mathsf{ls}(x, A, y) \stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge emp) \vee (x \neq y \wedge \exists z, v, A'.\ A = v :: A' \wedge \mathsf{node}(x, v, z) * \mathsf{ls}(z, A', y))$$
$$\mathsf{ls}(x, y) \stackrel{\text{def}}{=} \exists A.\ \mathsf{ls}(x, A, y)$$
$$\mathsf{garb} \stackrel{\text{def}}{=} \exists S_g.\ (\mathtt{GN} = S_g) * (\circledast_{x \in S_g} . \mathsf{node}(x))$$

$$R = G \stackrel{\text{def}}{=} (\mathit{Push} \vee \mathit{Pop} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$
$$\mathit{Push} \stackrel{\text{def}}{=} \exists x, y, v, A.\ ((\mathtt{Stk} = A) \wedge (\mathtt{S} = y)) \propto ((\mathtt{Stk} = v :: A) \wedge (\mathtt{S} = x) * \mathsf{node}(x, v, y))$$
$$\mathit{Pop} \stackrel{\text{def}}{=} \exists x, y, v, A, S_g.\ ((\mathtt{Stk} = v :: A) \wedge (\mathtt{S} = x) * \mathsf{node}(x, v, y) * (\mathtt{GN} = S_g)) \propto ((\mathtt{Stk} = A) \wedge (\mathtt{S} = y) * \mathsf{node}(x, v, y) * (\mathtt{GN} = S_g \cup \{x\}))$$

Figure 23: Precise invariant, rely and guarantee of Treiber stack.

The guarantee includes the push and the pop actions. At the concrete side, the steps at line `8` for `push` and line `12` for `pop` in Figure 22 are the linearization points, i.e., they correspond to the abstract atomic `PUSH` and `POP` operations (thus the effect bits of the actions are **true**!). Note that when popping a node, we also add the node to `GN`. The rely of a thread is the same as its guarantee.

We show the proof in Figure 24. For linearizability, we let the abstract operations be executed simultaneously with the concrete code at linearization points. Note that when popping from an empty stack, the linearization point is at line `5` (see `pop` in Figure 22), where the thread reads the stack pointer.

On lock-freedom, we know the failure of the `cas`es at line `8` for `push` and line `12` for `pop` must be caused by the successful progress of other threads. In the proof, we can increase the number of tokens when the environment updates the `S` pointer (i.e., the environment does *Push* or *Pop*), thus are allowed to do more loop iterations.

```
 1 push(v) {
 2   local x, t, b;
     {I ∧ arem(PUSH(V)) ∧ v = V}
 3   b := false;
 4   x := cons(v, null);
     {(¬b ∧ I ∗ node(x, v, _) ∧ arem(PUSH(V)) ∧ (v = V))
      ∨ (b ∧ I ∧ arem(skip))}        //Applying the WHILE rule and the HIDE-W rule
 5   while (!b) {
       {¬b ∧ I ∗ node(x, v, _) ∧ arem(PUSH(V)) ∧ (v = V) ∧ wf(0)}
 6     < t := S; >
 7     x.next := t;
       {¬b ∧ I ∗ node(x, v, t) ∧ arem(PUSH(V)) ∧ (v = V)
        ∧ ∃a. (S = a) ∗ true ∧ (t = a ∧ wf(0) ∨ t ≠ a ∧ wf(1))}
 8     b := cas(&S, t, x);
       {(b ∧ I ∧ arem(skip) ∧ wf(1))
        ∨ (¬b ∧ I ∗ node(x, v, _) ∧ arem(PUSH(V)) ∧ (v = V) ∧ wf(1))}
 9   }
     {I ∧ arem(skip)}
10 }

IntSet GN;
//Auxiliary global variable for verification: popped garbage nodes

 1 pop() {
 2   local v, x, t, b;
     {I ∧ arem(POP)}
 3   b := false;
     {(¬b ∧ I ∧ arem(POP)) ∨ (b ∧ I ∧ arem(skip) ∧ (v = V))}
              //Applying the WHILE rule and the HIDE-W rule
 4   while (!b) {
       {¬b ∧ I ∧ arem(POP) ∧ wf(0)}
 5     < t := S; >
       {(t = null ∧ ¬b ∧ I ∧ arem(skip) ∧ (V = EMPTY) ∧ wf(1))
        ∨ (¬b ∧ I ∧ arem(POP) ∧ ∃a. (S = a) ∗ node(t) ∗ true ∧ (t = a ∧ wf(0) ∨ t ≠ a ∧ wf(1)))}
 6     if (t = null) {
         {t = null ∧ ¬b ∧ I ∧ arem(skip) ∧ (V = EMPTY)}
 7       v := EMPTY;
 8       b := true;
         {b ∧ I ∧ arem(skip) ∧ (v = V = EMPTY)}
 9     } else {
         {¬b ∧ I ∧ arem(POP) ∧ ∃a. (S = a) ∗ node(t) ∗ true ∧ (t = a ∧ wf(0) ∨ t ≠ a ∧ wf(1))}
10       v := t.data;
11       x := t.next;
         {¬b ∧ I ∧ arem(POP) ∧ ∃a. (S = a) ∗ node(t, v, x) ∗ true ∧ (t = a ∧ wf(0) ∨ t ≠ a ∧ wf(1))}
12       < b := cas(&S, t, x); GN := GN ∪ {t}; >
         {(b ∧ I ∧ arem(skip) ∧ (v = V) ∧ wf(1)) ∨ (¬b ∧ I ∧ arem(POP) ∧ wf(1))}
13     }
14   }
     {I ∧ arem(skip) ∧ (v = V)}
15   return v;
16 }
```

Figure 24: Proof outline for Treiber stack.

4.4 MS Lock-Free Queue

```
 1 enq(v) {
 2  local x, t, s, b;
 3  b := false;
 4  x := cons(v, null);
 5  while (!b) {
 6    < t := Tail; >
 7    s := t.next;
 8    if (t = Tail) {
 9      if (s = null) {
10        b := cas(&(t.next), s, x);
11        if (b) {
12          cas(&Tail, t, x);
13        }
14      } else {
15        cas(&Tail, t, s);
16      }
17    }
18  }
19 }
```

```
 1 deq() {
 2  local v, s, h, t, b;
 3  b := false;
 4  while (!b) {
 5    < h := Head; >
 6    < t := Tail; >
 7    s := h.next;
 8    if (h = t) {
 9      if (s = null) {
10        v := EMPTY;
11        b := true;
12      } else {
13        cas(&Tail, t, s);
14      }
15    } else {
16      v := s.val;
17      b := cas(&Head, h, s);
18    }
19  }
20  return v;
21 }
```

```
 1 ENQ(V) {
 2  < Q := Q :: V; >
 3 }

 1 DEQ() {
 2  local V;
 3  < if (Q = ϵ) {
 4      V := EMPTY;
 5    } else {
 6      V := head(Q);
 7      Q := tail(Q);
 8    }
 9  >
10  return V;
11 }
```

Figure 25: Variant of MS lock-free queue.

In Figure 25, we show a variant[2] of Michael-Scott lock-free queue [8] (at the left of the figure) and the abstract atomic operations (at the right). We use our logic to prove the linearizability and lock-freedom together of the MS queue. By similar arguments as for Treiber stack in Section 4.3, here we only need to prove the following:

$$R, G, I \vdash \{I \wedge \mathsf{arem}(\texttt{ENQ(V)}) \wedge (\texttt{v} = \texttt{V})\}\ \texttt{enq(v)}\ \{I \wedge \mathsf{arem}(\boldsymbol{skip})\}$$
$$\text{and} \qquad R, G, I \vdash \{I \wedge \mathsf{arem}(\texttt{DEQ})\}\ \texttt{deq}\ \{I \wedge \mathsf{arem}(\boldsymbol{skip}) \wedge (\texttt{v} = \texttt{V})\}$$

We define the precise invariant I, the rely R and the guarantee G in Figure 26, and show the proof in Figures 27 and 28, The invariant I for the well-formed shared data structure is defined in the same way as in linearizability proofs (e.g., [6]). Here we introduce an auxiliary variable `GH` to collect those nodes which were dequeued from the list. Initially it is set to `Head`, and would not change any more. Then the list segment from `GH` to `Head` includes all the dequeued nodes.

The rely R and the guarantee G contain three actions in addition to identity transitions: *Enq*, *Deq* and *Swing*. The actions *Enq* and *Deq* insert and remove a node from the queue, and correspond to abstract steps (the effect bits are **true**). The action *Swing* moves the `Tail` pointer, which does not correspond to any abstract steps.

The proofs in Figures 27 and 28 are based on the linearizability proofs (e.g., [6]) but also take into account the lock-freedom property.[3] We need to specify in the loop invariants (in both Figures 27 and 28)

[2]We removed in `deq` the double check on the read of the `Head` pointer. As explained in our previous work [6], this double check introduces a non-fixed linearization point in this queue algorithm, but removing it would not affect the correctness of the algorithm. Currently we use a simplified setting and do not support non-fixed linearization points (since they are orthogonal to our main focus in this paper on *termination preservation*). We can further extend the logic in this paper with the techniques for verifying linearizability with non-fixed linearization points [6], then we would be able to verify the original MS queue implementation. Due to the same reason, we remove the double check in DGLM queue implementation as well.

[3]We actually found that the lock-freedom proofs in Hoffmann et al's work [5] has bugs on computing the number of tokens. The authors confirmed our finding in our private communications.

$$I \stackrel{\text{def}}{=} \exists h, t, A.\ (\mathtt{Q} = A) \wedge (\mathtt{Head} = h) * (\mathtt{Tail} = t) * \mathsf{lsq}(h, t, A) * \mathsf{garb}(h)$$

$$\mathsf{node}(x, v, y) \stackrel{\text{def}}{=} x \mapsto (v, y) \qquad \mathsf{node}(x, y) \stackrel{\text{def}}{=} \mathsf{node}(x, \_, y) \qquad \mathsf{garb}(h) \stackrel{\text{def}}{=} \exists g.\ (\mathtt{GH} = g) * \mathsf{ls}(g, h)$$

$$\mathsf{lsq}(h, t, A) \stackrel{\text{def}}{=} \exists v, A', A''.\ (v :: A = A' :: A'') \wedge \mathsf{ls}(h, A', t) * \mathsf{tls}(t, \_, A'')$$

$$\mathsf{ls}(x, A, y) \stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge emp) \vee (x \neq y \wedge \exists z, v, A'.\ A = v :: A' \wedge \mathsf{node}(x, v, z) * \mathsf{ls}(z, A', y))$$

$$\mathsf{ls}(x, y) \stackrel{\text{def}}{=} \exists A.\ \mathsf{ls}(x, A, y)$$

$$\mathsf{last2}(t, v, x, v') \stackrel{\text{def}}{=} \mathsf{node}(t, v, x) * \mathsf{node}(x, v', \mathtt{null}) \qquad \mathsf{last2}(t, x) \stackrel{\text{def}}{=} \mathsf{last2}(t, \_, x, \_) \qquad \mathsf{last2}(t) \stackrel{\text{def}}{=} \mathsf{last2}(t, \_)$$

$$\mathsf{tls}(t, x, A) \stackrel{\text{def}}{=} \exists v, v'.\ (A = v \wedge \mathsf{node}(t, v, x) \wedge x = \mathtt{null}) \vee (A = v :: v' \wedge \mathsf{last2}(t, v, x, v')) \qquad \mathsf{tls}(t, x) \stackrel{\text{def}}{=} \exists A.\ \mathsf{tls}(t, x, A)$$

$$R = G \stackrel{\text{def}}{=} (\mathit{Enq} \vee \mathit{Deq} \vee \mathit{Swing} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

$$\mathit{Enq} \stackrel{\text{def}}{=} \exists v, v', A, t, x.\ ((\mathtt{Q} = A) \wedge (\mathtt{Tail} = t) * \mathsf{node}(t, v, \mathtt{null})) \propto ((\mathtt{Q} = A :: v') \wedge (\mathtt{Tail} = t) * \mathsf{last2}(t, v, x, v'))$$

$$\mathit{Deq} \stackrel{\text{def}}{=} \exists v, A, h, t, x, y.\ ((\mathtt{Q} = v :: A) \wedge (\mathtt{Head} = h) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y) * (\mathtt{Tail} = t) \wedge h \neq t) \propto ((\mathtt{Q} = A) \wedge (\mathtt{Head} = x) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y) * (\mathtt{Tail} = t))$$

$$\mathit{Swing} \stackrel{\text{def}}{=} \exists v, v', t, x.\ (\boldsymbol{emp} \wedge (\mathtt{Tail} = t) * \mathsf{last2}(t, v, x, v')) \ltimes (\boldsymbol{emp} \wedge (\mathtt{Tail} = x) * \mathsf{last2}(t, v, x, v'))$$

Figure 26: Precise invariant, rely and guarantee of MS lock-free queue. The auxiliary global variable `GH` is set to `Head` in the initialization method.

the least number n of tokens to execute the loops (i.e., the thread can only run the loop for no more than n rounds before it or its environment fulfills some source steps). For instance, in the proof for `enq` (Figure 27), when the `Tail` pointer lags behind the last node, we need to have at least two tokens to first advance the `Tail` pointer in one iteration and then enqueue a node in another iteration. Thus we define tw (in Figure 27) saying that we have at least two tokens if `Tail` lags behind and one token otherwise. It is part of our loop invariants in both the proofs for `enq` and `deq`. Moreover, to maintain this loop invariant, we should get *two* more tokens whenever the environment enqueues a node (such that the `Tail` pointer lags behind the last node) and makes the `cas` of the current thread fail.

$\mathsf{tw}(t) \stackrel{\text{def}}{=} (\mathtt{Tail} = t) * ((\mathsf{last2}(t) \wedge \mathsf{wf}(2)) \vee (\mathsf{node}(t, \mathtt{null}) \wedge \mathsf{wf}(1)))$ $\qquad \mathsf{tw} \stackrel{\text{def}}{=} \exists t.\ \mathsf{tw}(t)$

$\mathsf{tw'}(t, n) \stackrel{\text{def}}{=} (\mathtt{Tail} = t) * ((\mathsf{last2}(t, n) \wedge \mathsf{wf}(1)) \vee (\mathsf{node}(t, n) \wedge n = \mathtt{null} \wedge \mathsf{wf}(0)))$

$\mathsf{tw'}(t) \stackrel{\text{def}}{=} \mathsf{tw'}(t, \_)$ $\qquad \mathsf{tw'} \stackrel{\text{def}}{=} \exists t.\ \mathsf{tw'}(t)$

$\mathsf{newTail}(n) \stackrel{\text{def}}{=} (\mathsf{node}(n, \mathtt{null}) * (\mathtt{Tail} = n) \wedge \mathsf{wf}(1)) \vee (\mathsf{last2}(n) * (\mathtt{Tail} = n) \wedge \mathsf{wf}(2))$
$\vee (\exists x, y.\ \mathsf{node}(n, x) * \mathsf{ls}(x, y) * \mathsf{tw}(y) \wedge \mathsf{wf}(2))$

$\mathsf{readTailEnvAdv}(t, n) \stackrel{\text{def}}{=} \mathsf{node}(t, n) * \mathsf{newTail}(n)$ $\qquad \mathsf{readTailEnvAdv}(t) \stackrel{\text{def}}{=} \mathsf{readTailEnvAdv}(t, \_)$

$\mathsf{readTail}(t) \stackrel{\text{def}}{=} \mathsf{tw'}(t) \vee \mathsf{readTailEnvAdv}(t)$

$\mathsf{readTailNextNullEnv}(t, n) \stackrel{\text{def}}{=} (n = \mathtt{null}) \wedge ((\mathtt{Tail} = t) * \mathsf{last2}(t) \wedge \mathsf{wf}(2)) \vee \mathsf{readTailEnvAdv}(t))$

$\mathsf{readTailNext}(t, n) \stackrel{\text{def}}{=} \mathsf{tw'}(t, n) \vee \mathsf{readTailEnvAdv}(t, n) \vee \mathsf{readTailNextNullEnv}(t, n)$

$\mathsf{readTailNextNull}(t, n) \stackrel{\text{def}}{=} ((\mathtt{Tail} = t) * \mathsf{node}(t, n) \wedge n = \mathtt{null} \wedge \mathsf{wf}(0)) \vee \mathsf{readTailNextNullEnv}(t, n)$

$\mathsf{readTailNextNonnull}(t, n) \stackrel{\text{def}}{=} ((\mathtt{Tail} = t) * \mathsf{last2}(t, n) \wedge \mathsf{wf}(1)) \vee \mathsf{readTailEnvAdv}(t, n)$

```
1  enq(v) {
2   local x, t, s, b;
    {I ∧ arem(ENQ(V)) ∧ v = V}
3   b := false;
4   x := cons(v, null);
    {(¬b ∧ I * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V))
     ∨ (b ∧ I ∧ arem(skip))}        //Applying the WHILE rule and the HIDE-W rule
5   while (!b) {
      {¬b ∧ (I ∧ tw' * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
6     < t := Tail; >
      {¬b ∧ (I ∧ readTail(t) * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
7     s := t.next;
      {¬b ∧ (I ∧ readTailNext(t, s) * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
8     if (t = Tail) {
        {¬b ∧ (I ∧ readTailNext(t, s) * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
9       if (s = null) {
          {¬b ∧ (I ∧ readTailNextNull(t, s) * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
10        b := cas(&(t.next), s, x);
          {(b ∧ I ∧ readTailNextNonnull(t, x) * true ∧ arem(skip))
           ∨ (¬b ∧ (I ∧ readTailNextNullEnv(t, s) * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V))}
11        if (b) {
            {b ∧ I ∧ readTailNextNonnull(t, x) * true ∧ arem(skip)}
12          cas(&Tail, t, x);
            {b ∧ I ∧ arem(skip)}
13        }
          {(b ∧ I ∧ arem(skip))
           ∨ (¬b ∧ (I ∧ tw * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V))}
14      } else {
          {¬b ∧ (I ∧ readTailNextNonnull(t, s) * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
15        cas(&Tail, t, s);
          {¬b ∧ (I ∧ tw * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V)}
16      }
17    }
      {(¬b ∧ (I ∧ tw * true) * node(x, v, null) ∧ arem(ENQ(V)) ∧ (v = V))
       ∨ (b ∧ I ∧ arem(skip))}
18  }
    {I ∧ arem(skip)}
19 }
```

Figure 27: Proof outline for enq of MS lock-free queue.

$\mathsf{readHeadEnv}(h,n,x) \stackrel{\text{def}}{=} (h \neq x) \wedge \mathsf{node}(h,n) * \mathsf{ls}(n,x) * (\mathtt{Head} = x)$

$\mathsf{readHead}(h,x) \stackrel{\text{def}}{=} ((h = x) \wedge (\mathtt{Head} = x)) \vee (\mathsf{readHeadEnv}(h,\_,x) * \mathsf{wf}(1))$ $\quad$ $\mathsf{readHead}(h) \stackrel{\text{def}}{=} \mathsf{readHead}(h,\_)$

$\mathsf{readTailAfterHead}(h,t) \stackrel{\text{def}}{=} \exists x.\ \mathsf{readHead}(h,x) * \mathsf{ls}(x,t) * \mathsf{readTail}(t)$

$\mathsf{readHeadNextAfterTail}(h,n,t) \stackrel{\text{def}}{=} (((\mathtt{Head} = h) \wedge (h = t)) * \mathsf{readTailNext}(t,n))$
$\vee ((\mathtt{Head} = h) * \mathsf{node}(h,n) * \mathsf{ls}(n,t) * \mathsf{readTail}(t))$
$\vee (\exists x.\ \mathsf{readHeadEnv}(h,n,x) * \mathsf{wf}(1) * \mathsf{ls}(x,t) * \mathsf{readTail}(t))$

$\mathsf{readHeadNextVal}(h,n,v) \stackrel{\text{def}}{=} ((\mathtt{Head} = h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,\_) * (\mathtt{Tail} = n))$
$\vee (\exists x,t.\ (\mathtt{Head} = h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,x) * \mathsf{ls}(x,t) * (\mathtt{Tail} = t))$
$\vee (\mathsf{readHeadEnv}(h,n,\_) * \mathsf{tw})$

```
1 deq() {
2  local v, s, h, t, b;
   {I ∧ arem(DEQ)}
3  b := false;
   {(¬b ∧ I ∧ arem(DEQ)) ∨ (b ∧ I ∧ arem(skip) ∧ (v = V))}
            //Applying the WHILE rule and the HIDE-W rule
4  while (!b) {
     {¬b ∧ I ∧ tw' * true ∧ arem(DEQ)}
5    < h := Head; >
     {¬b ∧ I ∧ tw' * readHead(h) * true ∧ arem(DEQ)}
6    < t := Tail; >
     {¬b ∧ I ∧ readTailAfterHead(h, t) * true ∧ arem(DEQ)}
7    s := h.next;
     {¬b ∧ I ∧ readHeadNextAfterTail(h, s, t) * true
      ∧ ((h = t ∧ s = null ∧ arem(skip) ∧ V = EMPTY) ∨ ((h ≠ t ∨ s ≠ null) ∧ arem(DEQ)))}
8    if (h = t) {
9      if (s = null) {
         {¬b ∧ I ∧ h = t ∧ s = null ∧ arem(skip) ∧ V = EMPTY}
10       v := EMPTY;
11       b := true;
         {b ∧ I ∧ arem(skip) ∧ (v = V = EMPTY)}
12     } else {
         {¬b ∧ I ∧ readHeadNextAfterTail(h, s, t) * true ∧ h = t ∧ s ≠ null ∧ arem(DEQ)}
         {¬b ∧ I ∧ readTailNextNonnull(t, s) * true ∧ arem(DEQ)}
13       cas(&Tail, t, s);
         {¬b ∧ I ∧ tw * true ∧ arem(DEQ)}
14     }
15   } else {
       {¬b ∧ I ∧ readHeadNextAfterTail(h, s, t) * true ∧ h ≠ t ∧ arem(DEQ)}
16     v := s.val;
       {¬b ∧ I ∧ readHeadNextAfterTail(h, s, t) * true ∧ node(s, v, _) * true ∧ h ≠ t ∧ arem(DEQ)}
       {¬b ∧ I ∧ readHeadNextVal(h, s, v) * true ∧ arem(DEQ)}
17     < b := cas(&Head, h, s); >
       {(¬b ∧ I ∧ tw * true ∧ arem(DEQ)) ∨ (b ∧ I ∧ arem(skip) ∧ (v = V))}
18   }
     {(¬b ∧ I ∧ tw * true ∧ arem(DEQ)) ∨ (b ∧ I ∧ arem(skip) ∧ (v = V))}
19 }
   {I ∧ arem(skip) ∧ (v = V)}
20 return v;
21 }
```

Figure 28: Proof outline for a variant of deq in MS lock-free queue.

4.5 DGLM Lock-Free Queue

```
 1 enq(v) {
 2  local x, t, s, b;
 3  b := false;
 4  x := cons(v, null);
 5  while (!b) {
 6    < t := Tail; >
 7    s := t.next;
 8    if (t = Tail) {
 9      if (s = null) {
10        b := cas(&(t.next), s, x);
11        if (b) {
12          cas(&Tail, t, x);
13        }
14      } else {
15        cas(&Tail, t, s);
16      }
17    }
18  }
19 }
```

```
 1 deq() {
 2  local v, s, h, t, b;
 3  b := false;
 4  while (!b) {
 5    < h := Head; >
 6    s := h.next;
 7    if (s = null) {
 8      v := EMPTY;
 9      b := true;
10    } else {
11      v := s.val;
12      b := cas(&Head, h, s);
13      if (b) {
14        < t := Tail; >
15        if (h = t) {
16          cas(&Tail, t, s);
17        }
18      }
19    }
20  }
21  return v;
22 }
```

Figure 29: Variant of DGLM lock-free queue.

Doherty et al. [1] present an optimized version of the `deq` method in MS lock-free queue, and verify linearizability of the algorithm by constructing a forward and a backward simulations. Here we prove its linearizability and lock-freedom together. We show a variant[4] of the code in Figure 29. Its `enq` method is the same as the MS lock-free queue. For `deq`, it tests whether `Tail` points to the sentinel node (line `15` in Figure 29) only after `Head` has been updated (line `12`), while in Michael and Scott's version, the test (line `8` in the `deq` of Figure 25) is performed before knowing the queue is not empty.

The precise invariant I and the rely/guarantee conditions R and G are almost the same as MS lock-free queue, as shown in Figure 30. The proof for `enq` is the same as that of MS lock-free queue. In Figure 31, we show the proof of the `deq` method for the DGLM queue using our logic. Different from the `deq` method of MS queue, here we would not first use one iteration to advance the `Tail` pointer before dequeuing nodes (instead, only after we have dequeued nodes, we may advance the `Tail` pointer, as shown at line `16` of the `deq` method in Figure 29). Thus in the loop invariant, we no longer need to have at least two tokens when `Tail` lags behind the last node. We can just use $\mathsf{wf}(1)$ as the loop invariant on the number of tokens, for all cases.

$$I \stackrel{\text{def}}{=} \exists h, t, A.\ (\&\mathtt{Q} \Mapsto A) \wedge (\&\mathtt{Head} \mapsto h) * (\&\mathtt{Tail} \mapsto t) * (\mathsf{lsq}(h, t, A) \vee \mathsf{cross}(h, t, A)) * \mathsf{garb}(h)$$
$$\mathsf{cross}(h, t, A) \stackrel{\text{def}}{=} (A = \epsilon) \wedge \mathsf{node}(t, h) * \mathsf{node}(h, \mathtt{null})$$

$$R = G \stackrel{\text{def}}{=} (\mathit{Enq} \vee \mathit{Deq} \vee \mathit{Swing} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$
$$\mathit{Deq} \stackrel{\text{def}}{=} \exists v, A, h, x, y.\ ((\&\mathtt{Q} \Mapsto v :: A) \wedge (\&\mathtt{Head} \mapsto h) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y))$$
$$\propto\ ((\&\mathtt{Q} \Mapsto A) \wedge (\&\mathtt{Head} \mapsto x) * \mathsf{node}(h, x) * \mathsf{node}(x, v, y))$$

Figure 30: Precise invariant, rely and guarantee of DGLM lock-free queue. Here lsq, garb, *Enq* and *Swing* are the same as those for MS queue.

[4] As for MS lock-free queue, we also remove the double check on the read of `Head` in the `deq` method of DGLM queue.

$\mathsf{readHeadNextNullEnv}(h,n) \stackrel{\text{def}}{=} (n = \mathtt{null}) \wedge \exists x, y.\ \mathsf{node}(h,x) * ((\mathsf{node}(x,y) * (\&\mathtt{Head} \mapsto h)) \vee (\mathsf{ls}(x,y) * (\&\mathtt{Head} \mapsto y)))$

$\mathsf{readHeadNext}(h,n) \stackrel{\text{def}}{=} (\mathsf{node}(h,n) * (\&\mathtt{Head} \mapsto h)) \vee (\mathsf{readHeadEnv}(h,n,x) * \mathsf{wf}(1)) \vee \mathsf{readHeadNextNullEnv}(h,n)$

$\mathsf{readHeadNextVal}(h,n,v) \stackrel{\text{def}}{=} ((\&\mathtt{Head} \mapsto h) * \mathsf{node}(h,n) * \mathsf{node}(n,v,\_)) \vee (\mathsf{readHeadEnv}(h,n,x) * \mathsf{wf}(1))$

$\mathsf{readTailEnvAdv}(t,n) \stackrel{\text{def}}{=} \exists x.\ (x \neq t) \wedge \mathsf{node}(t,n) * \mathsf{ls}(n,x) * (\&\mathtt{Tail} \mapsto x)$

$\mathsf{readTail}(t) \stackrel{\text{def}}{=} ((\&\mathtt{Tail} \mapsto t) * \mathsf{tls}(t,\_)) \vee \mathsf{readTailEnvAdv}(t,\_)$

$\mathsf{readLagTail}(t,n) \stackrel{\text{def}}{=} ((\&\mathtt{Tail} \mapsto t) * \mathsf{last2}(t,n)) \vee \mathsf{readTailEnvAdv}(t,n)$

```
1  deq() {
2    local v, s, h, t, b;
     {I ∧ arem(DEQ)}
3    b := false;
     {(¬b ∧ I ∧ arem(DEQ)) ∨ (b ∧ I ∧ arem(skip) ∧ (v = V))}
                //Applying the WHILE rule and the HIDE-W rule
4    while (!b) {
       {¬b ∧ I ∧ arem(DEQ) ∧ wf(0)}
5      < h := Head; >
       {¬b ∧ I ∧ readHead(h) * true ∧ arem(DEQ)}
6      s := h.next;
       {¬b ∧ I ∧ readHeadNext(h, s) * true
        ∧ ((s = null ∧ arem(skip) ∧ V = EMPTY) ∨ (s ≠ null ∧ arem(DEQ)))}
7      if (s = null) {
         {¬b ∧ I ∧ s = null ∧ arem(skip) ∧ V = EMPTY}
8        v := EMPTY;
9        b := true;
         {b ∧ I ∧ arem(skip) ∧ (v = V = EMPTY)}
10     } else {
         {¬b ∧ I ∧ readHeadNext(h, s) * true ∧ (s ≠ null) ∧ arem(DEQ))}
11       v := s.val;
         {¬b ∧ I ∧ readHeadNextVal(h, s, v) * true ∧ arem(DEQ))}
12       b := cas(&Head, h, s);
         {(b ∧ I ∧ node(h, s) * node(s, _) * true ∧ arem(skip) ∧ (v = V)) ∨ (¬b ∧ I ∧ arem(DEQ) ∧ wf(1))}
13       if (b) {
           {b ∧ I ∧ node(h, s) * node(s, _) * true ∧ arem(skip) ∧ (v = V)}
14         < t := Tail; >
           {b ∧ I ∧ node(h, s) * node(s, _) * true ∧ readTail(t) * true ∧ arem(skip) ∧ (v = V)}
15         if (h = t) {
             {b ∧ I ∧ readLagTail(t, s) * true ∧ arem(skip) ∧ (v = V)}
16           cas(&Tail, t, s);
17         }
           {b ∧ I ∧ arem(skip) ∧ (v = V)}
18       }
19     }
       {(¬b ∧ I ∧ arem(DEQ) ∧ wf(1)) ∨ (b ∧ I ∧ arem(skip) ∧ (v = V))}
20   }
     {I ∧ arem(skip) ∧ (v = V)}
21   return v;
22 }
```

Figure 31: Proof outline for a variant of deq in DGLM lock-free queue. Here readHead and readHeadEnv are the same as those for MS queue.

4.6 Synchronous Queue

```
 1 initialize() {
 2   local sentinel;
 3   sentinel := new Node(null, DATA, null);
 4   GH := Head := Tail := sentinel;
 5 }
```

```
 1 enq(v) {
 2  local t, h, n, offer, b, v';
 3  b := false;
 4  offer := new Node(v, DATA, null);
 5  while (!b) {
 6    t := Tail;
 7    h := Head;
 8    if (h = t || t.type = DATA) {
 9      n := t.next;
10      if (t = Tail) {
11        if (n != null) {
12          cas(&Tail, t, n);
13        } else if (cas(&(t.next), n, offer)){
14          cas(&Tail, t, offer);
15          v' := offer.data;
16          while (v' = v) { v' := offer.data; }
17          h := Head;
18          if (offer = h.next)
19            cas(&Head, h, offer);
20          b := true;
21        }
22      }
23    } else {
24      n := h.next;
25      if (t = Tail && h = Head && n != null) {
26        b := cas(&(n.data), null, v);
27        cas(Head, h, n);
28        if (b) free(offer);
29      }
30    }
31  }
32 }
```

```
 1 deq() {
 2  local t, h, n, req, b, v;
 3  b := false;
 4  req := new Node(null, REQ, null);
 5  while (!b) {
 6    t := Tail;
 7    h := Head;
 8    if (h = t || t.type = REQ) {
 9      n := t.next;
10      if (t = Tail) {
11        if (n != null) {
12          cas(&Tail, t, n);
13        } else if (cas(&(t.next), n, req)){
14          cas(&Tail, t, req);
15          v := req.data;
16          while (v = null) { v := req.data; }
17          h := Head;
18          if (req = h.next)
19            cas(&Head, h, req);
20          b := true;
21        }
22      }
23    } else {
24      n := h.next;
25      if (t = Tail && h = Head && n != null) {
26        v := n.data;
27        if (v != null) {
28          b := cas(&(n.data), v, null);
29        }
30        cas(Head, h, n);
31        if (b) free(offer);
32      }
33    }
34  }
35  return v;
36 }
```

Figure 32: Synchronous dual queue. Here `GH` is an auxiliary variable.

A synchronous queue is a concurrent transfer channel in which each producer presenting an item must wait for a consumer to take this item, and vice versa. We show the implementation of synchronous queue (used in Java 6 [9]) in Figure 32. It is based on the Michael-Scott queue. At any time, the queue contains either `enq` reservations (nodes whose `type` fields are `DATA`), `deq` reservations (nodes whose `type` fields are `REQ`), or it is empty. In the `enq` method (also known as `put`), a thread first checks if the queue is empty or contains `DATA`-type reservations (line 8 in `enq` in Figure 32). If so, it enqueues (puts in) its new `DATA`-type reservation (lines 13 and 14 in `enq`), and waits at the item for a `deq` thread to take it (lines 15 and 16 in `enq`). When a `deq` thread finds this reservation, it will take away the data contained in the item (line 26 in `deq`), set the `data` field to `null` (line 28 in `deq`) and remove this item (line 30 in `deq`). Also when the waiting `enq` thread finds that the item has been taken, it can try to remove the item as well (lines 18 and 19 in `enq`). Symmetrically, a `deq` thread first checks if the queue is empty or contains

REQ-type reservations (line 8 in deq), and if so, it enqueues (puts in) its new REQ-type reservation (lines 13 and 14 in deq), and waits for a enq thread to fulfill it (lines 15 and 16 in deq).

The synchronous queue does not satisfy the traditional linearizability definition [4]. But we can see that the steps for a thread to put in its reservation (which are actually like the enq method in MS queue in Figure 25) are "linearizable" and "lock-free" (in that the multiple steps can be abstracted as an atomic operation), and the steps for taking away the data or fulfilling the reservation (which are like the deq method in MS queue) are also "linearizable" and "lock-free". The waiting steps are certainly not "lock-free" which require interactions from other threads to progress. We can define non-atomic abstract code and prove that the synchronous queue implementation refines it.

```
 1 ENQ(V) {
 2  local nd, mustWait, va;
 3  < nd := dequeue(D);
 4    mustWait := (nd = null);
 5    if (mustWait) { nd := enqueue(E, V); }
 6  >
 7  if (mustWait) {
 8    va := nd.data;
 9    while(va = V) { va := nd.data; }
10  }
11  else {
12    nd.data := V;
13  }
14 }
```

```
 1 DEQ() {
 2  local nd, mustWait, V;
 3  < nd := dequeue(E);
 4    mustWait := (nd = null);
 5    if (mustWait) { nd := enqueue(D, null); }
 6  >
 7  if (mustWait) {
 8    V := nd.data;
 9    while(V = null) { V := nd.data; }
10  }
11  else {
12    V := nd.data;
13    nd.data := null;
14  }
15  return V;
16 }
```

Figure 33: Abstract synchronous queue.

As shown in Figure 33, the abstract code follows Java SE 5.0 SynchronousQueue class [9]. We maintain two abstract queues: D for waiting dequeuers and E for waiting enqueuers. Each queue is a mathematical list of node *addresses* (as an abstraction/simplification of a linked list). The command enqueue(E, v) allocates a new abstract node with data v and inserts its address at the tail of the queue E, and returns the address. The command dequeue(E) removes the first item (a node address) from the queue E and returns it if E is not empty ($E \neq \epsilon$), and returns null otherwise.

In the ENQ method, a thread first checks if D is empty (line 4 of ENQ in Figure 33), and if so, it atomically puts in its reservation to E (line 5). Then it waits for a deq thread to take away the data in the reservation (lines 8 and 9). If D is not empty, then it dequeues a reservation from D and writes its enqueued value V to the data field of the reservation (line 12). The DEQ method is symmetric.

To simplify the proof, we assume the abstract state always contain a dummy node whose data is null. The node is never accessed by the code. It is used to correspond to the initial sentinel node of the concrete list.

To prove the concrete implementation in Figure 32 refines the abstract operations in Figure 33 using our logic, we first define the invariant I and the rely and guarantee conditions R and G in Figure 34.

The invariant I says, the shared memory contains the queue Q and some garbage nodes Garb which were removed from the queue by either enq or deq. As usual we introduce an auxiliary variable GH to collect those nodes which were removed from the list. Initially it is set to Head, and would not change any more. Then the list segment (Gls) from GH to Head includes all the removed nodes. Also these removed nodes must have been sentinel nodes (stnl), i.e., those DATA-type nodes whose data has been taken and those REQ-type nodes whose data has been fulfilled. The queue Q is either a DATA-type queue (and the abstract D must be empty) or a REQ-type queue (and the abstract E must be empty). And it always contains one or two sentinel nodes (the two-sentinel case occurs since the Head pointer may lag behind

the new sentinel node). Also as in MS queue, the `Tail` pointer may lag behind the last node. But if `Head` lags behind the new sentinel node, `Tail` would not be equal to `Head`, as indicated by the implementation in Figure 32.

The rely and guarantee conditions contain six possible actions in addition to the identity transitions. *AdvHead* and *AdvTail* are to swing the `Head` and `Tail` pointers when they lag behind. These two actions do not correspond to any abstract step. *ResvE* and *ResvD* each inserts a new node at the tail of the queue. *Put* fulfills the `data` field of a `REQ`-type node at the head of the queue, and *Take* takes away the `data` of a `DATA`-type node. They both make a normal node into a sentinel node. The four actions *ResvE*, *ResvD*, *Put* and *Take* correspond to abstract steps and thus their effect bits must be **true**.

We show the proofs of `enq` in Figures 37 and 38, with some auxiliary predicates defined in Figures 35 and 36. Proofs for `deq` is symmetric and omitted here. Similar to the proofs for MS queue, we need to specify in the loop invariants the least number n of tokens to execute the loops (i.e., the thread can only run the loop for no more than n rounds before it or its environment fulfills some source steps). In the proof for `enq` (Figure 37), when either the `Head` or the `Tail` pointer lags behind, we need to have at least two tokens (as defined by loopInv in Figure 35). To maintain this loop invariant, we should get *two* more tokens whenever the environment inserts a node at the tail (such that the `Tail` pointer lags behind the last node), and whenever the environment makes a normal node becomes a sentinel node (such that the `Head` pointer lags behind the new sentinel).

$$I \stackrel{\text{def}}{=} \exists h, t.\ (\mathtt{Head} \doteq h) * (\mathtt{Tail} \doteq t) * \mathsf{Q}(h,t) * \mathsf{Garb}(h)$$

$$\mathsf{Q}(h,t) \stackrel{\text{def}}{=} \exists b.\ \mathsf{Q}_b(h,t)$$

$$\mathsf{Q}_b(h,t) \stackrel{\text{def}}{=} \exists L.\ \mathsf{Q}_b(h,t,L) * ((b = \mathtt{DATA} \wedge (\mathtt{E} \doteq L) * (\mathtt{D} \doteq \epsilon)) \vee (b = \mathtt{REQ} \wedge (\mathtt{D} \doteq L) * (\mathtt{E} \doteq \epsilon)))$$

$$\begin{aligned}\mathsf{Q}_b(h,t,L) \stackrel{\text{def}}{=}\ & \mathsf{Ss}_b(h,t,\mathtt{null}) \wedge L = \epsilon \\ & \vee \exists x, X.\ \mathsf{Ss}_b(h,t,x) * \mathsf{Qn}_b(x,\mathtt{null},X) \wedge L = X :: \epsilon \\ & \vee \exists x, L', L''.\ \mathsf{Ss}_b(h,\_,x) * \mathsf{Qls}_b(x,t,L') * \mathsf{Qtl}_b(t,\_,L'') \wedge L = L' :: L''\end{aligned}$$

$$\mathsf{Garb}(h) \stackrel{\text{def}}{=} \exists g.\ (\mathtt{GH} \doteq g) * \mathsf{Gls}(g,h)$$

$$\mathsf{Ss}(x,y,z) \stackrel{\text{def}}{=} \exists b.\ \mathsf{Ss}_b(x,y,z) \qquad \mathsf{Ss}_b(x,y,z) \stackrel{\text{def}}{=} (\mathsf{Stnl}(x,z) \wedge (x = y)) \vee (\mathsf{Stnl}(x,y) * \mathsf{Stnl}_b(y,z))$$

$$\mathsf{Gls}(x,y) \stackrel{\text{def}}{=} (x = y) \vee (x \neq y \wedge \exists z.\ \mathsf{Stnl}(x,z) * \mathsf{Gls}(z,y))$$

$$\mathsf{Qls}_b(x,y,L) \stackrel{\text{def}}{=} (x = y \wedge L = \epsilon) \vee (x \neq y \wedge \exists z, X, L'.\ L = X :: L' \wedge \mathsf{Qn}_b(x,z,X) * \mathsf{Qls}_b(z,y,L'))$$

$$\begin{aligned}\mathsf{Qtl}_b(x,y,L) \stackrel{\text{def}}{=}\ & (\exists X.\ \mathsf{Qn}_b(x,y,X) \wedge y = \mathtt{null} \wedge L = X :: \epsilon) \\ & \vee (\exists X, Y.\ \mathsf{Qn}_b(x,y,X) * \mathsf{Qn}_b(y,\mathtt{null},Y) \wedge L = X :: Y :: \epsilon)\end{aligned}$$

$$\mathsf{Stnl}(x,y) \stackrel{\text{def}}{=} \exists b.\ \mathsf{Stnl}_b(x,\_,y,\_) \qquad \mathsf{Stnl}_b(x,v,y,X) \stackrel{\text{def}}{=} \mathsf{stnl}_b(x,v,y) \wedge \mathsf{NODE}(X,v)$$

$$\mathsf{Qn}_b(x,y,X) \stackrel{\text{def}}{=} \mathsf{Qn}_b(x,\_,y,X) \qquad \mathsf{Qn}_b(x,v,y,X) \stackrel{\text{def}}{=} \mathsf{qn}_b(x,v,y) \wedge \mathsf{NODE}(X,v)$$

$$\mathsf{stnl}_b(x,v,y) \stackrel{\text{def}}{=} \mathsf{node}_b(x,v,y) \wedge ((b = \mathtt{DATA} \wedge v = \mathtt{null}) \vee (b = \mathtt{REQ} \wedge v \neq \mathtt{null}))$$

$$\mathsf{qn}_b(x,v,y) \stackrel{\text{def}}{=} \mathsf{node}_b(x,v,y) \wedge ((b = \mathtt{DATA} \wedge v \neq \mathtt{null}) \vee (b = \mathtt{REQ} \wedge v = \mathtt{null}))$$

$$\mathsf{node}_b(x,v,y) \stackrel{\text{def}}{=} x \mapsto (v,b,y) \qquad \mathsf{NODE}(X,V) \stackrel{\text{def}}{=} X \Mapsto (V)$$

$$\mathsf{stnl}(x,y) \stackrel{\text{def}}{=} \exists b.\ \mathsf{stnl}_b(x,\_,y) \qquad \mathsf{qn}(x,y) \stackrel{\text{def}}{=} \exists b.\ \mathsf{qn}_b(x,\_,y) \qquad \mathsf{node}(x,y) \stackrel{\text{def}}{=} \exists b.\ \mathsf{node}_b(x,\_,y)$$

$$\mathsf{stnl}_b(x,y) \stackrel{\text{def}}{=} \mathsf{stnl}_b(x,\_,y) \qquad \mathsf{node}_b(x,y) \stackrel{\text{def}}{=} \mathsf{node}_b(x,\_,y) \qquad \mathsf{node}(x,v,y) \stackrel{\text{def}}{=} \exists b.\ \mathsf{node}_b(x,v,y)$$

$$R = G \stackrel{\text{def}}{=} (\mathit{AdvHead} \vee \mathit{AdvTail} \vee \mathit{ResvE} \vee \mathit{ResvD} \vee \mathit{Put} \vee \mathit{Take} \vee \mathsf{Id}) * \mathsf{Id} \wedge (I \ltimes I)$$

$$\mathit{AdvHead} \stackrel{\text{def}}{=} \exists x, y, z, s.\ [\mathsf{stnl}(x,y) * \mathsf{stnl}(y,z) \wedge \boldsymbol{emp}] * ((\mathtt{Head} \doteq x) \ltimes (\mathtt{Head} \doteq y))$$

$$\mathit{AdvTail} \stackrel{\text{def}}{=} \exists x, y.\ [\mathsf{node}(x,y) * \mathsf{node}(y,\mathtt{null}) \wedge \boldsymbol{emp}] * ((\mathtt{Tail} \doteq x) \ltimes (\mathtt{Tail} \doteq y))$$

$$\begin{aligned}\mathit{ResvE} \stackrel{\text{def}}{=}\ & \exists v, v', b, t, x, L, X.\ ((\mathtt{Tail} = t) * \mathsf{node}_b(t,v,\mathtt{null}) \wedge (\mathtt{E} = L) * (\mathtt{D} = \epsilon)) \\ & \propto ((\mathtt{Tail} = t) * \mathsf{node}_b(t,v,x) * \mathsf{qn}_{\mathtt{DATA}}(x,v',\mathtt{null}) \wedge (\mathsf{NODE}(X,v') * (\mathtt{E} = L :: X) * (\mathtt{D} = \epsilon)))\end{aligned}$$

$$\begin{aligned}\mathit{ResvD} \stackrel{\text{def}}{=}\ & \exists v, v', b, t, x, L, X.\ ((\mathtt{Tail} = t) * \mathsf{node}_b(t,v,\mathtt{null}) \wedge (\mathtt{E} = \epsilon) * (\mathtt{D} = L)) \\ & \propto ((\mathtt{Tail} = t) * \mathsf{node}_b(t,v,x) * \mathsf{qn}_{\mathtt{REQ}}(x,v',\mathtt{null}) \wedge (\mathsf{NODE}(X,v') * (\mathtt{E} = \epsilon) * (\mathtt{D} = L :: X)))\end{aligned}$$

$$\begin{aligned}\mathit{Put} \stackrel{\text{def}}{=}\ & \exists h, t, x, y, X, L.\ [(\mathtt{Head} \doteq h) * (\mathtt{Tail} \doteq t) * \mathsf{Stnl}(h,x) * (\mathtt{E} \doteq \epsilon) \wedge (h \neq t)] \\ & * ((\mathsf{Qn}_{\mathtt{REQ}}(x,y,X) * (\mathtt{D} \doteq X :: L)) \propto (\mathsf{Stnl}_{\mathtt{REQ}}(x,y,X) * (\mathtt{D} \doteq L))\end{aligned}$$

$$\begin{aligned}\mathit{Take} \stackrel{\text{def}}{=}\ & \exists h, t, x, y, X, L.\ [(\mathtt{Head} \doteq h) * (\mathtt{Tail} \doteq t) * \mathsf{Stnl}(h,x) * (\mathtt{D} \doteq \epsilon) \wedge (h \neq t)] \\ & * ((\mathsf{Qn}_{\mathtt{DATA}}(x,y,X) * (\mathtt{E} \doteq X :: L)) \propto (\mathsf{Stnl}_{\mathtt{DATA}}(x,y,X) * (\mathtt{E} \doteq L))\end{aligned}$$

Figure 34: Precise invariant, rely and guarantee of synchronous queue. Here we use $E_1 \doteq E_2$ and $\mathbb{E}_1 \doteq \mathbb{E}_2$ short for $(E_1 = E_2) \wedge \boldsymbol{emp}$ and $(\mathbb{E}_1 = \mathbb{E}_2) \wedge emp$ respectively.

$\mathsf{node2}_p(t,n,x) \stackrel{\text{def}}{=} \mathsf{node}_p(t,n) * \mathsf{node}(n,x) \qquad \mathsf{node2}(t,n,x) \stackrel{\text{def}}{=} \exists p.\ \mathsf{node2}_p(t,n,x)$

$\mathsf{stnl2}_p(h,n,v) \stackrel{\text{def}}{=} \mathsf{stnl}(h,n) * \mathsf{stnl}_p(n,v,\_) \qquad \mathsf{stnl2}_p(h) \stackrel{\text{def}}{=} \mathsf{stnl2}_p(h,\_,\_) \qquad \mathsf{stnl2}(h) \stackrel{\text{def}}{=} \exists p.\ \mathsf{stnl2}_p(h)$

$\mathsf{stnl1}_p(h,n,v) \stackrel{\text{def}}{=} \mathsf{stnl}(h,n) * \mathsf{qn}_p(n,v,\_) \qquad \mathsf{stnl1}_p(h) \stackrel{\text{def}}{=} \mathsf{stnl1}_p(h,\_,\_) \qquad \mathsf{stnl1}(h) \stackrel{\text{def}}{=} \exists p.\ \mathsf{stnl1}_p(h)$

$\mathsf{gls}(x,y) \stackrel{\text{def}}{=} (x=y) \vee (x \neq y \wedge \exists z.\ \mathsf{stnl}(x,z) * \mathsf{gls}(z,y))$

$\mathsf{ls}(x,y) \stackrel{\text{def}}{=} (x=y) \vee (x \neq y \wedge \exists z.\ \mathsf{node}(x,z) * \mathsf{ls}(z,y))$

$\mathsf{lagTail} \stackrel{\text{def}}{=} \mathsf{node2}(\mathtt{Tail},\_,\mathtt{null}) \qquad \mathsf{nonlagTail} \stackrel{\text{def}}{=} \mathsf{node}(\mathtt{Tail},\mathtt{null}) \qquad \mathsf{tail} \stackrel{\text{def}}{=} \mathsf{lagTail} \vee \mathsf{nonlagTail}$

$\mathsf{lagHead} \stackrel{\text{def}}{=} \mathsf{stnl2}(\mathtt{Head}) \qquad \mathsf{nonlagHead} \stackrel{\text{def}}{=} \mathsf{stnl}(\mathtt{Head},\mathtt{null}) \vee \mathsf{stnl1}(\mathtt{Head}) \qquad \mathsf{head} \stackrel{\text{def}}{=} \mathsf{lagHead} \vee \mathsf{nonlagHead}$

$\mathsf{loopInv} \stackrel{\text{def}}{=} ((\mathsf{lagTail} \vee \mathsf{lagHead}) \wedge \mathsf{wf}(2)) \vee (\mathsf{nonlagTail} \wedge \mathsf{nonlagHead} \wedge \mathsf{wf}(1))$

$\mathsf{loopBody} \stackrel{\text{def}}{=} ((\mathsf{lagTail} \vee \mathsf{lagHead}) \wedge \mathsf{wf}(1)) \vee (\mathsf{nonlagTail} \wedge \mathsf{nonlagHead} \wedge \mathsf{wf}(0))$

$$\begin{aligned}\mathsf{newTail}_p(n,v) \stackrel{\text{def}}{=}\ & (\mathsf{node}_p(n,v,\mathtt{null}) \wedge (n=\mathtt{Tail}) \wedge \mathsf{wf}(1)) \\ & \vee (\exists x.\ \mathsf{node}_p(n,v,x) * \mathsf{node}(x,\mathtt{null}) \wedge (n=\mathtt{Tail}) \wedge \mathsf{wf}(2)) \\ & \vee (\exists x.\ \mathsf{node}_p(n,v,x) * \mathsf{ls}(x,\mathtt{Tail}) * \mathsf{tail} \wedge \mathsf{wf}(2))\end{aligned}$$

$\mathsf{newTail}(n) \stackrel{\text{def}}{=} \exists p,v.\ \mathsf{newTail}_p(n,v) \qquad \mathsf{NewTail}_p(n,v,N) \stackrel{\text{def}}{=} \mathsf{newTail}_p(n,v) * \mathsf{NODE}(N,v)$

$\mathsf{readTailEnvAdv}_{p,q}(t,n,v) \stackrel{\text{def}}{=} \mathsf{node}_p(t,n) * \mathsf{newTail}_q(n,v)$

$\mathsf{readTailEnvAdv}_p(t) \stackrel{\text{def}}{=} \exists q.\ \mathsf{readTailEnvAdv}_{p,q}(t,\_,\_) \qquad \mathsf{readTailEnvAdv}_p(t,n) \stackrel{\text{def}}{=} \exists q.\ \mathsf{readTailEnvAdv}_{p,q}(t,n,\_)$

$\mathsf{readTail}_p(t) \stackrel{\text{def}}{=} (t=\mathtt{Tail} \wedge (\mathsf{node2}_p(t,\_,\mathtt{null}) \vee \mathsf{node}_p(t,\mathtt{null}))) \vee \mathsf{readTailEnvAdv}_p(t)$

$\mathsf{readTailNextNullEnv}_p(t,n) \stackrel{\text{def}}{=} (n=\mathtt{null}) \wedge ((t=\mathtt{Tail} \wedge \mathsf{node2}_p(t,\_,\mathtt{null}) \wedge \mathsf{wf}(2)) \vee \mathsf{readTailEnvAdv}_p(t))$

$$\begin{aligned}\mathsf{readTailNext}_p(t,n) \stackrel{\text{def}}{=}\ & (t=\mathtt{Tail} \wedge (\mathsf{node2}_p(t,n,\mathtt{null}) \vee (\mathsf{node}_p(t,n) \wedge n=\mathtt{null}))) \\ & \vee \mathsf{readTailEnvAdv}_p(t,n) \vee \mathsf{readTailNextNullEnv}_p(t,n)\end{aligned}$$

$\mathsf{readTailNextNonnull}_p(t,n) \stackrel{\text{def}}{=} (t=\mathtt{Tail} \wedge \mathsf{node2}_p(t,n,\mathtt{null}) \wedge \mathsf{wf}(1)) \vee \mathsf{readTailEnvAdv}_p(t,n)$

$\mathsf{readTailNextNull}_p(t,n) \stackrel{\text{def}}{=} (t=\mathtt{Tail} \wedge \mathsf{node}_p(t,n) \wedge n=\mathtt{null} \wedge \mathsf{wf}(0)) \vee \mathsf{readTailNextNullEnv}_p(t,n)$

$\mathsf{EnvXchg}_q(n,v,N) \stackrel{\text{def}}{=} \exists x.\ \mathsf{Stnl}_q(n,v,x,N) * \mathsf{ls}(x,\mathtt{Tail}) * \mathsf{tail} \wedge (\mathsf{stnl}(\mathtt{Head},n) \vee \mathsf{gls}(n,\mathtt{Head}))$

$\mathsf{EnvXchgReadHead}_q(n,v,N,h) \stackrel{\text{def}}{=} \exists x.\ \mathsf{Stnl}_q(n,v,x,N) * \mathsf{ls}(x,\mathtt{Tail}) * \mathsf{tail} \wedge (\mathsf{stnl}(h,n) \vee \mathsf{gls}(n,h)) \wedge \mathsf{gls}(h,\mathtt{Head})$

$\mathsf{EnvXchgLagHead}_q(n,v,N,h) \stackrel{\text{def}}{=} \exists x.\ \mathsf{Stnl}_q(n,v,x,N) * \mathsf{ls}(x,\mathtt{Tail}) * \mathsf{tail} \wedge \mathsf{stnl}(h,n) \wedge \mathsf{gls}(h,\mathtt{Head})$

$\mathsf{EnvXchgNonlagHead}_q(n,v,N) \stackrel{\text{def}}{=} \exists x.\ \mathsf{Stnl}_q(n,v,x,N) * \mathsf{ls}(x,\mathtt{Tail}) * \mathsf{tail} \wedge \mathsf{gls}(n,\mathtt{Head})$

$$\begin{aligned}\mathsf{Resv}_q(t,n,v,v',N) \stackrel{\text{def}}{=}\ & (t=\mathtt{Tail} \wedge \mathsf{node}(t,n) * \mathsf{Qn}_q(n,v,\mathtt{null},N)) \\ & \vee \mathsf{node}(t,n) * \mathsf{NewTail}_q(n,v,N) \vee \mathsf{node}(t,n) * \mathsf{EnvXchg}_q(n,v',N)\end{aligned}$$

$\mathsf{ResvAdv}_q(n,v,v',N) \stackrel{\text{def}}{=} \mathsf{NewTail}_q(n,v,N) \vee \mathsf{EnvXchg}_q(n,v',N)$

$\mathsf{ResvAdvReadData}_q(n,v,v',v_r,N) \stackrel{\text{def}}{=} \mathsf{NewTail}_q(n,v,N) \wedge (v_r=v) \vee \mathsf{EnvXchg}_q(n,v',N) \wedge (v_r=v' \vee v_r=v)$

$\mathtt{ENQWait} \stackrel{\text{def}}{=} (\mathtt{va := nd.data;\ ENQWhile})$

$\mathtt{ENQWhile} \stackrel{\text{def}}{=} (\mathtt{while(va=V)\{\ va := nd.data;\ \}})$

Figure 35: Auxiliary definition - I.

$$\mathsf{newHead}_p(n,v) \stackrel{\text{def}}{=} (\mathsf{stnl}_p(n,v,\mathtt{null}) \wedge (n = \mathtt{Head}) \wedge \mathsf{wf}(1)) \\ \vee (\exists x.\ \mathsf{stnl}_p(n,v,x) * \mathsf{qn}(x,\_) \wedge (n = \mathtt{Head}) \wedge \mathsf{wf}(1)) \\ \vee (\exists x.\ \mathsf{stnl}_p(n,v,x) * \mathsf{stnl}(x,\_) \wedge (n = \mathtt{Head}) \wedge \mathsf{wf}(2)) \\ \vee (\exists x.\ \mathsf{stnl}_p(n,v,x) * \mathsf{gls}(x,\mathtt{Head}) * \mathsf{head} \wedge \mathsf{wf}(2))$$

$$\mathsf{newHead}(n) \stackrel{\text{def}}{=} \exists p, v.\ \mathsf{newHead}_p(n,v)$$

$$\mathsf{readHeadEnvAdv}_p(h,n,v) \stackrel{\text{def}}{=} \mathsf{stnl}(h,n) * \mathsf{newHead}_p(n,v)$$

$$\mathsf{readHeadEnvAdv}_p(h) \stackrel{\text{def}}{=} \mathsf{readHeadEnvAdv}_p(h,\_,\_) \qquad \mathsf{readHeadEnvAdv}_p(h,n) \stackrel{\text{def}}{=} \mathsf{readHeadEnvAdv}_p(h,n,\_)$$

$$\mathsf{readHead}_p(h) \stackrel{\text{def}}{=} (h = \mathtt{Head} \wedge (\mathsf{stnl}_p(h,\mathtt{null}) \vee \mathsf{stnl1}_p(h) \vee \mathsf{stnl2}_p(h))) \ \vee\ \mathsf{readHeadEnvAdv}_p(h)$$

$$\mathsf{readHeadNextNullEnv}_p(h,n) \stackrel{\text{def}}{=} (n = \mathtt{null}) \wedge ((h = \mathtt{Head} \wedge \mathsf{stnl}_p(h,x) * \mathsf{node}(x,\_) \wedge \mathsf{wf}(2)) \vee \mathsf{readHeadEnvAdv}_p(h))$$

$$\mathsf{readHeadNext}_p(h,n) \stackrel{\text{def}}{=} (h = \mathtt{Head} \wedge ((\mathsf{stnl}_p(h,n) \wedge n = \mathtt{null}) \vee \mathsf{stnl1}_p(h,n,\_) \vee \mathsf{stnl2}_p(h,n,\_))) \\ \vee \mathsf{readHeadEnvAdv}_p(h,n) \ \vee\ \mathsf{readHeadNextNullEnv}_p(h,n)$$

$$\mathsf{readHeadNextNonnull}_p(h,n) \stackrel{\text{def}}{=} (h = \mathtt{Head} \wedge (\mathsf{stnl1}_p(h,n,\_) \vee \mathsf{stnl2}_p(h,n,\_))) \ \vee\ \mathsf{readHeadEnvAdv}_p(h,n)$$

$$\mathsf{readHeadNextNull}_p(h,n) \stackrel{\text{def}}{=} (h = \mathtt{Head} \wedge \mathsf{stnl}_p(h,n) \wedge n = \mathtt{null}) \ \vee\ \mathsf{readHeadNextNullEnv}_p(h,n)$$

$$\mathsf{Xchg}_p(h,n,v) \stackrel{\text{def}}{=} (h = \mathtt{Head} \wedge \mathsf{stnl2}_p(h,n,v)) \ \vee\ \mathsf{readHeadEnvAdv}_p(h,n,v)$$

$$\mathsf{Xchg}_p(h,n) \stackrel{\text{def}}{=} \mathsf{Xchg}_p(h,n,\_)$$

Figure 36: Auxiliary definition - II.

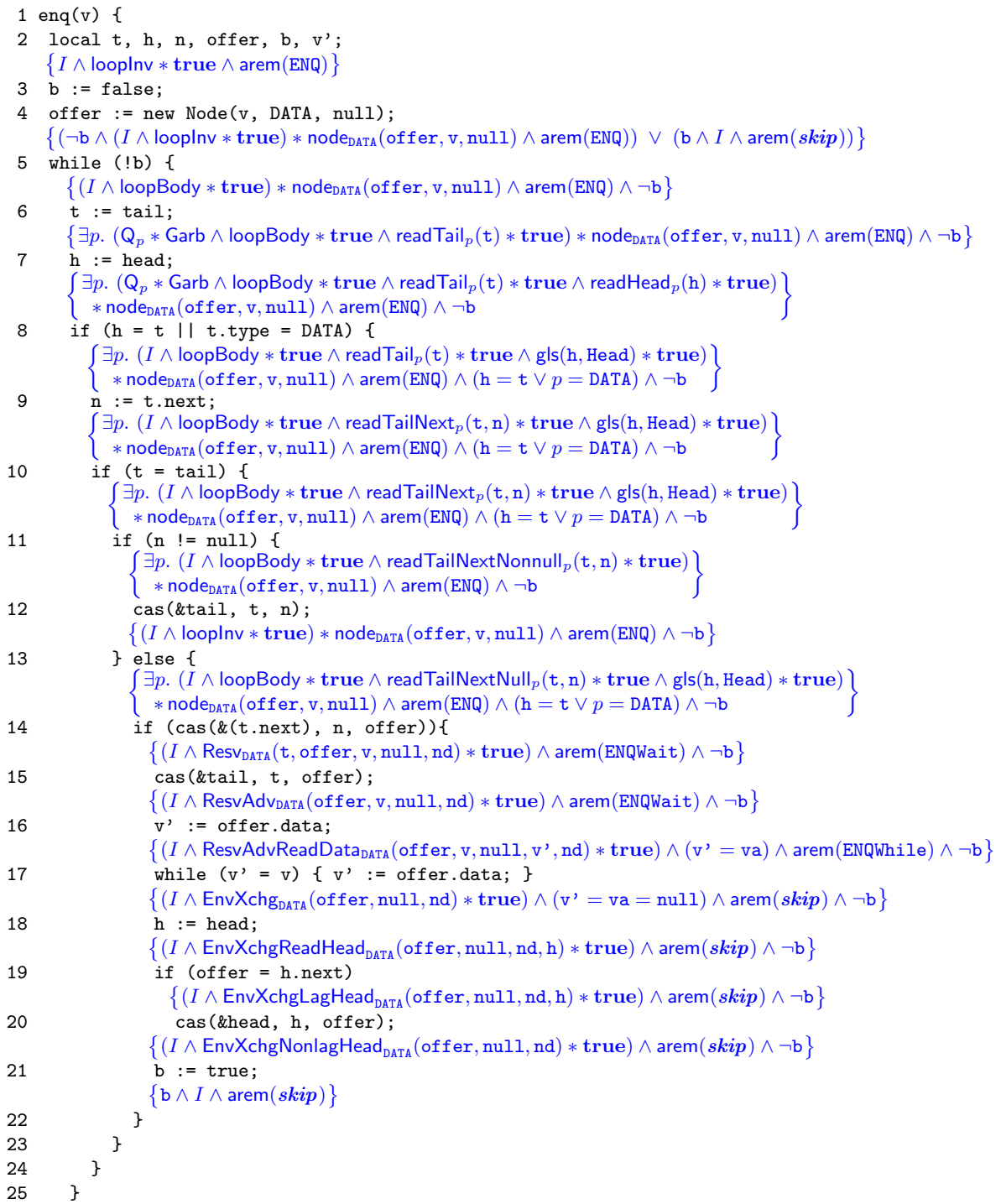


Figure 37: Proof outline - I.

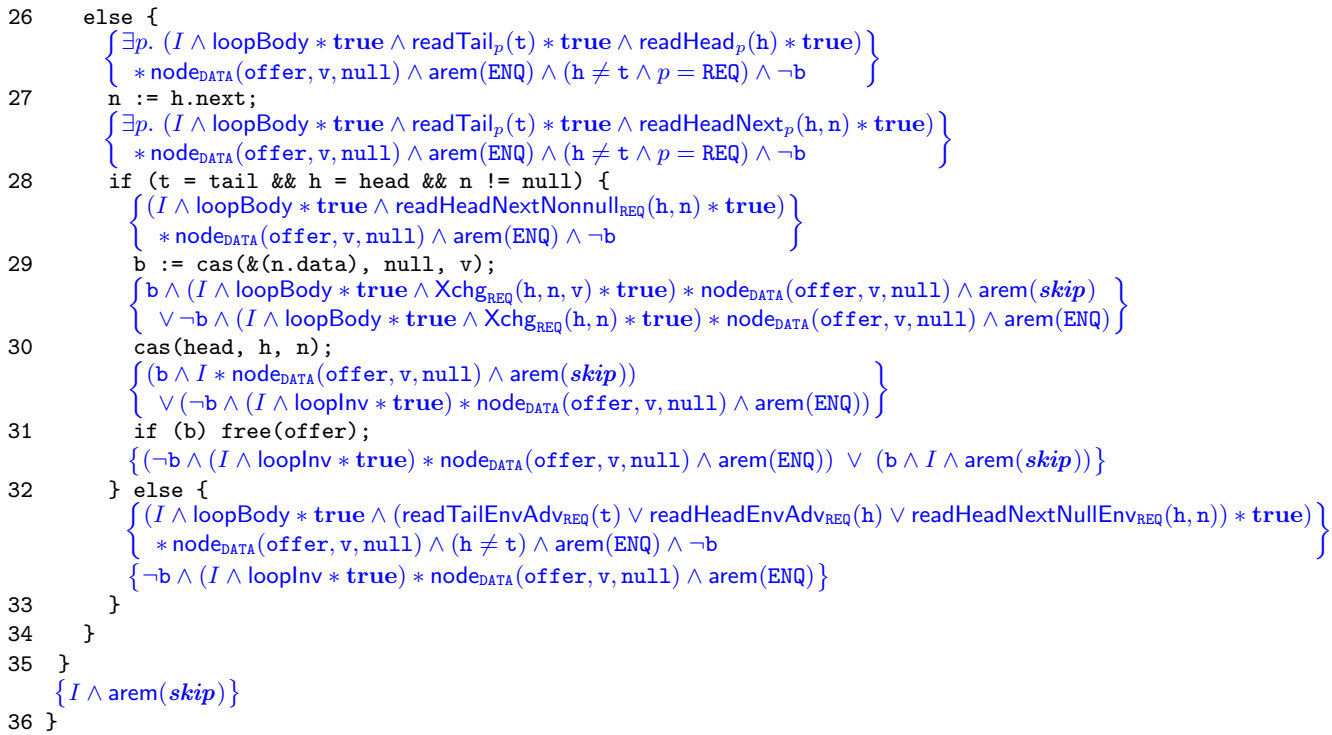


Figure 38: Proof outline - II.

5 Soundness Proofs

Below we first prove the adequacy of RGSim-T w.r.t. the termination-sensitive refinement (Section 5.1). Then we define the unary judgment semantics (Section 5.2), and we prove the soundness of the binary inference rules of Figure 7 (Section 5.3), where the binary judgment semantics is just RGSim-T in Definition 2, and also prove the soundness of the unary rules of Figure 8 (Section 5.4). Finally we show the derivation of the WHILE-TERM rule (Section 5.5).

5.1 Adequacy of RGSim-T

RGSim-T in Definition 2 (which is also the binary judgment semantics) implies the termination-sensitive refinement in Definition 1.

Theorem 4 (Adequacy of RGSim-T). If there exist R, G, I, Q and a metric M such that $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, then $(C, \sigma) \sqsubseteq (\mathbb{C}, \Sigma)$.

Proof: We want to prove the following: for any R, G, I, Q,

$$\begin{array}{l}\forall \mathbb{C}, \Sigma, \mathcal{E}.\\ (\,\exists C, \sigma, M.\ R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)\ \wedge\ ETr(C, \sigma, \mathcal{E})\,)\ \Longrightarrow\ ETr(\mathbb{C}, \Sigma, \mathcal{E})\end{array}$$

By co-induction.

$$\text{Co-induction Principle:}\ \ \forall x.\ \ (\exists S.\ S \subseteq F(S) \wedge x \in S)\ \Longrightarrow\ x \in \text{gfp}\ F$$

Figure 3 defines F and gfp F (i.e., ETr). Let

$$S \stackrel{\text{def}}{=} \{(\mathbb{C}, \Sigma, \mathcal{E}) \mid \exists C, \sigma, M.\ R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)\ \wedge\ ETr(C, \sigma, \mathcal{E})\}.$$

So from the co-induction principle, we only need to prove:

$$S \subseteq F(S),\ \ \text{i.e.,}\ \ \ \forall \mathbb{C}, \Sigma, \mathcal{E}.\ (\mathbb{C}, \Sigma, \mathcal{E}) \in S\ \Longrightarrow\ (\mathbb{C}, \Sigma, \mathcal{E}) \in F(S)\,.$$

After unfolding S, we only need to prove:

$$\forall M, \mathbb{C}, \Sigma, \mathcal{E}, C, \sigma.\ R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)\ \wedge\ ETr(C, \sigma, \mathcal{E})\ \Longrightarrow\ (\mathbb{C}, \Sigma, \mathcal{E}) \in F(S)\,. \tag{5.1}$$

By transfinite induction over M.

$$\text{Transfinite Induction Principle:}\ (\forall M.\ (\forall M'.\ M' < M \Longrightarrow P(M')) \Longrightarrow P(M)\,)\ \Longrightarrow\ \forall M.P(M)$$

We view (5.1) as $\forall M.P(M)$. So we only need to prove:

$$\begin{array}{l}\forall M.\\ \quad (\forall M'.\ M' < M\\ \quad\ \Longrightarrow\ (\forall \mathbb{C}', \Sigma', \mathcal{E}', C', \sigma'.\ R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')\ \wedge\ ETr(C', \sigma', \mathcal{E}')\\ \quad\quad\ \Longrightarrow\ (\mathbb{C}', \Sigma', \mathcal{E}') \in F(S))\,)\\ \Longrightarrow\\ \quad (\forall \mathbb{C}, \Sigma, \mathcal{E}, C, \sigma.\ R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)\ \wedge\ ETr(C, \sigma, \mathcal{E})\\ \quad\ \Longrightarrow\ (\mathbb{C}, \Sigma, \mathcal{E}) \in F(S))\end{array}$$

By inversion over $ETr(C, \sigma, \mathcal{E})$,

1. $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$ and $\mathcal{E} = \Downarrow$:

   From $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, we know there exists Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^* (\boldsymbol{skip}, \Sigma')$.

   Thus from the definition of F (Figure 3), we know $(\mathbb{C}, \Sigma, \mathcal{E}) \in F(S)$.

2. $(C, \sigma) \longrightarrow^+ \textbf{abort}$ and $\mathcal{E} = \lightning$:
   From $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, we know $(\mathbb{C}, \Sigma) \longrightarrow^+ \textbf{abort}$.
   Thus from the definition of F (Figure 3), we know $(\mathbb{C}, \Sigma, \mathcal{E}) \in F(S)$.

3. $(C, \sigma) \longrightarrow^+ (C', \sigma')$ and $ETr(C', \sigma', \mathcal{E})$:
   From $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, we know one of the following two cases holds:
   (a) there exist M', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma) \longrightarrow^+ (\mathbb{C}', \Sigma')$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$.
   Thus $(\mathbb{C}', \Sigma', \mathcal{E}) \in S$. Then from the definition of F (Figure 3), we know $(\mathbb{C}, \Sigma, \mathcal{E}) \in F(S)$.
   (b) there exists M' such that $M' < M$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}, \Sigma)$.
   Then from the induction hypothesis, we know $ETr(\mathbb{C}, \Sigma, \mathcal{E})$.

4. $(C, \sigma) \xrightarrow{e}{}^+ (C', \sigma')$, $ETr(C', \sigma', \mathcal{E}')$ and $\mathcal{E} = e :: \mathcal{E}'$:
   From $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, we know:
   there exist $\mathbb{C}'$, Σ' and M' such that $(\mathbb{C}, \Sigma) \xrightarrow{e}{}^+ (\mathbb{C}', \Sigma')$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$.
   Thus $(\mathbb{C}', \Sigma', \mathcal{E}') \in S$. Then from the definition of F (Figure 3), we know $(\mathbb{C}, \Sigma, \mathcal{E}) \in F(S)$.

Then we are done. □

5.2 Unary Judgment Semantics

The unary judgment semantics $R, G, I \models \{p\}C\{q\}$ follows RGSim-T (Definition 2). The initial abstract code in the simulation comes from the precondition p, and the postcondition q specifies the final abstract code that corresponds to the concrete final code **skip**. The assertions p and q also specify the while-specific metric w (the numbers of tokens), which must be related to the metric M used in the simulation RGSim-T.

Below we first show how we instantiate the abstract metric M in RGSim-T based on w.

5.2.1 Instantiation of the Abstract Metric M

For each single thread, its metric ws (defined below) is a list of (w, n) pairs, where w is the while-specific metric and n is "code size" which will be explained later. We let the threaded metric ws be a list (a stack actually) to allow different while-specific metrics for nested loops. That is, when entering a loop, we can push a (w, n) pair to the ws stack; and when exiting the loop, we pop the pair out of ws.

The threaded metric ws uses the dictionary order. However, the usual dictionary order over lists is not well-founded (consider $B > AB > AAB > AAAB > \ldots$ in a dictionary). To address this issue, we introduce a bound of the list length (stack height), $\mathcal{H}$, and define the well-founded order $<_{\mathcal{H}}$ by requiring the lists should be not longer than $\mathcal{H}$. Intuitively, the stack height $\mathcal{H}$ represents the maximal depth of nested loops, so it can be determined for any given program.

To get the whole-program metric, we compose threaded metrics by pairing them. Thus the abstract metric M in RGSim-T is instantiated as follows:

$$M \ ::= \ (ws, \mathcal{H}) \mid (M, M)$$

and we define the well-founded oder $<$ and the composition operation $+$ (see Lemma 16) as follows:

$$\frac{ws' <_{\mathcal{H}} ws \qquad \mathcal{H}' = \mathcal{H}}{(ws', \mathcal{H}') < (ws, \mathcal{H})} \qquad \frac{M_1' < M_1 \qquad M_2' = M_2}{(M_1', M_2') < (M_1, M_2)} \qquad \frac{M_1' = M_1 \qquad M_2' < M_2}{(M_1', M_2') < (M_1, M_2)}$$

$$M_1 + M_2 \stackrel{\text{def}}{=} (M_1, M_2)$$

The threaded metric ws and the well-founded order $<_{\mathcal{H}}$ are defined below. Note that we allow "$A < AB < B$" in a dictionary.

$$\begin{array}{rrcl} (\mathit{WfStack}) & ws & ::= & (w, n) \mid (w, n) :: ws \\ (\mathit{StkHeight}) & \mathcal{H} & \in & \mathit{Nat} \end{array}$$

$$ws' <_{\mathcal{H}} ws \quad \text{iff} \quad (ws' \ll ws) \wedge (|ws'| \leq \mathcal{H}) \wedge (|ws| \leq \mathcal{H})$$

$$\frac{(w', n') < (w, n)}{(w', n') \ll (w, n)} \qquad \frac{(w', n') \leq (w, n)}{(w', n') \ll (w, n) :: ws_1} \qquad \frac{(w', n') < (w, n)}{(w', n') :: ws_1' \ll (w, n)}$$

$$\frac{(w', n') < (w, n)}{(w', n') :: ws_1' \ll (w, n) :: ws_1} \qquad \frac{(w', n') = (w, n) \qquad ws_1' \ll ws_1}{(w', n') :: ws_1' \ll (w, n) :: ws_1}$$

Here $|ws|$ is the length of ws, which is defined as follows:

$$\begin{array}{rcl} |(w, n)| & = & 1 \\ |(w, n) :: ws| & = & 1 + |ws| \end{array}$$

The well-founded order over the (w, n) pairs is a usual dictionary order:

$$\begin{array}{lll} (w', n') < (w, n) & \text{iff} & (w' < w) \vee (w' = w \wedge n' < n) \\ (w', n') = (w, n) & \text{iff} & (w' = w) \wedge (n' = n) \\ (w', n') \leq (w, n) & \text{iff} & (w', n') < (w, n) \vee (w', n') = (w, n) \end{array}$$

Lemma 5 (Well-foundedness). The relation $M' < M$ defined above is a well-founded relation.

Proof: Easy to prove from Lemma 6. □

Lemma 6. The relation $ws' <_{\mathcal{H}} ws$ defined above is a well-founded relation.

Proof: Suppose there is an infinite descending chain:

$$ws_0 > ws_1 > ws_2 > \dots \tag{5.2}$$

Thus we know

$$ws_0 \gg ws_1 \gg ws_2 \gg \dots \tag{5.3}$$

and

$$\forall k.\ |ws_k| \leq \mathcal{H} \tag{5.4}$$

We prove the following property which generalizes (5.4) over the maximum size $\mathcal{H}$:

$$\forall ws_0, ws_1, ws_2, \dots.\ (\forall k.\ ws_k \gg ws_{k+1}) \implies (\forall m \geq 1.\ \exists j.\ |ws_j| > m) \tag{5.5}$$

By induction over m.

- **Base Case:** $m = 1$. Suppose $\forall k.\ |ws_k| = 1$. Thus we have an infinite descending chain:

  $$(w_0, n_0) > (w_1, n_1) > (w_2, n_2) > \dots \tag{5.6}$$

  It violates the definition of $(w', n') < (w, n)$ (which is a well-founded relation).

- **Inductive Step:** $m = m' + 1$. Since $(w', n') < (w, n)$ is a well-founded relation, we know there must exists k such that

  $$\forall j \geq k.\ \mathsf{root}(ws_j) = \mathsf{root}(ws_{j+1}) \tag{5.7}$$

  and there exist ws'_k, ws'_{k+1}, ws'_{k+2}, ... such that $\forall j \geq k.\ ws_j = \mathsf{root}(ws_j) :: ws'_j$ and

  $$\forall j \geq k.\ ws'_j \gg ws'_{j+1} \tag{5.8}$$

  Here $\mathsf{root}(ws)$ takes the first element of ws if ws has the first element and undefined otherwise. From the induction hypothesis, we know there exists $j \geq k$ such that

  $$|ws'_j| > m'. \tag{5.9}$$

  Thus $|ws_j| > m' + 1$.

So we are done. □

5.2.2 Intuitions of $\mathcal{H}$ and the Second Dimension of ws

Below we give more informal explanations (and examples) about the stack height $\mathcal{H}$ and the second dimension ("code size" n in each pair) of the threaded metric ws.

As we said, the stack height $\mathcal{H}$ represents the maximal depth of nested loops. For any given program C, we can determine the stack height using a function height defined in Figure 39.

The threaded metric ws as a stack requires us to distinguish the executions of the loop body from the executions of the code out of the loop. When entering a loop (for the first time), we can push a (w, n) pair onto the ws stack. But when we repeatedly execute the loop body (not for the first time), we do not want to push a new pair onto the stack.

Thus we introduce the runtime command $\mathsf{while}\ (B)\{C\}$ to represent the while-loop continuation when we have unfolded the loop **while** (B) C. And we revised the low-level operational semantics as follows:

$$
\begin{array}{rcl}
\mathsf{height}(\mathbf{skip}) & = & 1 \\
\mathsf{height}(c) & = & 1 \\
\mathsf{height}(\langle C \rangle) & = & 1 \\
\mathsf{height}(C_1; C_2) & = & max\{\mathsf{height}(C_1), \mathsf{height}(C_2)\} \\
\mathsf{height}(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2) & = & max\{\mathsf{height}(C_1), \mathsf{height}(C_2)\} \\
\mathsf{height}(\mathbf{while}\ (B)\ C) & = & \mathsf{height}(C) + 1
\end{array}
$$

Figure 39: Definition of height.

$$
\frac{[\![B]\!]_s = \mathbf{true}}{(\mathbf{while}\ (B)\ C, (s,h)) \longrightarrow (C; \mathsf{while}\ (B)\{C\}, (s,h))} \qquad \frac{[\![B]\!]_s = \mathbf{false}}{(\mathbf{while}\ (B)\ C, (s,h)) \longrightarrow (\mathbf{skip}, (s,h))}
$$

$$
\frac{[\![B]\!]_s = \mathbf{true}}{(\mathsf{while}\ (B)\{C\}, (s,h)) \longrightarrow (C; \mathsf{while}\ (B)\{C\}, (s,h))} \qquad \frac{[\![B]\!]_s = \mathbf{false}}{(\mathsf{while}\ (B)\{C\}, (s,h)) \longrightarrow (\mathbf{skip}, (s,h))}
$$

We can see that the new operational semantics for while loops is equivalent to the original one (see Figure 2). Below we will assume the new semantics and use it to prove the logic soundness. However, we want the readers to note that without the new operational semantics, we can still define the unary judgment semantics and prove the soundness of *all* the inference rules, based on the original operational semantics. The new operational semantics for while loops just makes the proofs (and the intuition) clearer, in particular, for the HIDE-W rule, the rule for "locally" reasoning about nested while loops.

With the runtime $\mathsf{while}\ (B)\{C\}$, we can calculate the code size n in each (w, n) pair of ws. We first label the code such that different layers of a nested while loop are assigned different labels.

Labeling the Code The syntax of the labeled code is defined below. Its operational semantics is straightforward, as shown in Figure 40.

$$
\begin{array}{rrcl}
(\mathit{Label}) & l & \in & \mathit{Nat} \\
(\mathit{LabStmt}) & \widehat{C} & ::= & \mathbf{skip}^l \mid c^l \mid \langle C \rangle^l \mid \widehat{C}_1; \widehat{C}_2 \mid \mathbf{if}^l(B)\ \widehat{C}_1\ \mathbf{else}\ \widehat{C}_2 \\
& & \mid & \mathbf{while}^l(B)\ \widehat{C} \mid \mathsf{while}^l(B)\ \widehat{C}
\end{array}
$$

We label the low-level code in the following way. Note that we do not need to label the runtime command $\mathsf{while}\ (B)\{C\}$, whose label is known during the runtime execution.

$$
\begin{array}{rcl}
\mathsf{labeling}(\mathbf{skip}, l) & = & \mathbf{skip}^l \\
\mathsf{labeling}(c, l) & = & c^l \\
\mathsf{labeling}(\langle C \rangle, l) & = & \langle C \rangle^l \\
\mathsf{labeling}(C_1; C_2, l) & = & \mathsf{labeling}(C_1, l); \mathsf{labeling}(C_2, l) \\
\mathsf{labeling}(\mathbf{if}\ (B)\ C_1\ \mathbf{else}\ C_2, l) & = & \mathbf{if}^l(B)\ \mathsf{labeling}(C_1, l)\ \mathbf{else}\ \mathsf{labeling}(C_2, l) \\
\mathsf{labeling}(\mathbf{while}\ (B)\ C, l) & = & \mathbf{while}^l(B)\ \mathsf{labeling}(C, l+1)
\end{array}
$$

We define the functions label, toplabel, minlabel and maxlabel in Figure 41. Then the stack height $\mathcal{H}$ of C is actually the maximum label of $\widehat{C}$, which is obtained by labeling C with 1. That is, the following holds:

$$
\mathsf{height}(C) \ = \ \mathsf{maxlabel}(\mathsf{labeling}(C, 1))
$$

We can prove the following property.

Lemma 7. For any C, $\widehat{C}$, $\widehat{C}'$, σ, σ' and R, if $\mathsf{labeling}(C, 1) = \widehat{C}$ and $(\widehat{C}, \sigma) \xmapsto{R}{}^* (\widehat{C}', \sigma')$, then there exist l, $\widehat{C}_1$, ..., $\widehat{C}_l$ such that $\widehat{C}' = (\widehat{C}_l; \ldots; \widehat{C}_1)$ and $\forall i \in [1..l].\ \mathsf{label}(\widehat{C}_i) = i$.

$$\frac{[\![B]\!]_s = \mathbf{true}}{(\mathbf{while}^l(B)\ \widehat{C}, (s,h)) \longrightarrow (\widehat{C}; \mathsf{while}^l(B)\ \widehat{C}, (s,h))} \qquad \frac{[\![B]\!]_s = \mathbf{false}}{(\mathbf{while}^l(B)\ \widehat{C}, (s,h)) \longrightarrow (\mathbf{skip}^l, (s,h))}$$

$$\frac{[\![B]\!]_s = \mathbf{true}}{(\mathsf{while}^l(B)\ \widehat{C}, (s,h)) \longrightarrow (\widehat{C}; \mathsf{while}^l(B)\ \widehat{C}, (s,h))} \qquad \frac{[\![B]\!]_s = \mathbf{false}}{(\mathsf{while}^l(B)\ \widehat{C}, (s,h)) \longrightarrow (\mathbf{skip}^l, (s,h))}$$

$$\frac{(\widehat{C}, \sigma) \longrightarrow (\widehat{C}', \sigma')}{(\widehat{C}; \widehat{C}'', \sigma) \longrightarrow (\widehat{C}'; \widehat{C}'', \sigma')} \qquad \frac{}{(\mathbf{skip}^l; \widehat{C}', \sigma) \longrightarrow (\widehat{C}', \sigma)}$$

$$\frac{(\widehat{C}, \sigma) \longrightarrow (\widehat{C}', \sigma')}{(\widehat{C}, \sigma) \overset{R}{\longmapsto} (\widehat{C}', \sigma')} \qquad \frac{((\sigma, \Sigma), (\sigma', \Sigma'), b) \models R}{(\widehat{C}, \sigma) \overset{R}{\longmapsto} (\widehat{C}, \sigma')}$$

Figure 40: Selected operational semantics rules of the labeled language.

$$\begin{aligned}
\mathsf{label}(\mathbf{skip}^l) &= l \\
\mathsf{label}(c^l) &= l \\
\mathsf{label}(\langle C \rangle^l) &= l \\
\mathsf{label}(\widehat{C}_1; \widehat{C}_2) &= \begin{cases} \mathsf{label}(\widehat{C}_1) & \text{if } \mathsf{label}(\widehat{C}_1) = \mathsf{label}(\widehat{C}_2) \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
\mathsf{label}(\mathbf{if}^l(B)\ \widehat{C}_1\ \mathbf{else}\ \widehat{C}_2) &= l \\
\mathsf{label}(\mathbf{while}^l(B)\ \widehat{C}) &= l \\
\mathsf{label}(\mathsf{while}^l(B)\ \widehat{C}) &= l
\end{aligned}$$

$$\begin{aligned}
\mathsf{minlabel}(\mathbf{skip}^l) &= l \\
\mathsf{minlabel}(c^l) &= l \\
\mathsf{minlabel}(\langle C \rangle^l) &= l \\
\mathsf{minlabel}(\widehat{C}_1; \widehat{C}_2) &= \mathsf{minlabel}(\widehat{C}_2) \\
\mathsf{minlabel}(\mathbf{if}^l(B)\ \widehat{C}_1\ \mathbf{else}\ \widehat{C}_2) &= l \\
\mathsf{minlabel}(\mathbf{while}^l(B)\ \widehat{C}) &= l \\
\mathsf{minlabel}(\mathsf{while}^l(B)\ \widehat{C}) &= l
\end{aligned}$$

$$\begin{aligned}
\mathsf{maxlabel}(\mathbf{skip}^l) &= l \\
\mathsf{maxlabel}(c^l) &= l \\
\mathsf{maxlabel}(\langle C \rangle^l) &= l \\
\mathsf{maxlabel}(\widehat{C}_1; \widehat{C}_2) &= max\{\mathsf{maxlabel}(\widehat{C}_1), \mathsf{maxlabel}(\widehat{C}_2)\} \\
\mathsf{maxlabel}(\mathbf{if}^l(B)\ \widehat{C}_1\ \mathbf{else}\ \widehat{C}_2) &= max\{\mathsf{maxlabel}(\widehat{C}_1), \mathsf{maxlabel}(\widehat{C}_2)\} \\
\mathsf{maxlabel}(\mathbf{while}^l(B)\ \widehat{C}) &= \mathsf{maxlabel}(\widehat{C})
\end{aligned}$$

Figure 41: Functions on labeled code.

It says, at any time in the execution of $\widehat{C}$, the runtime code must be in the form of $\widehat{C}_l; \widehat{C}_{l-1} \ldots; \widehat{C}_1$, where each $\widehat{C}_i$ has a fixed label i.

Code Sizes for Labeled Code For each pair (w, n) in any ws, n can be statically determined by the code. We use $\mathsf{proj}_2(ws)$ to project each pair (w, n) in ws to n. $\mathsf{proj}_1(ws)$ is defined similarly.

$$ns \quad ::= \quad n \mid n :: ns$$

$$\begin{array}{rcl} \mathsf{proj}_2(w, n) & = & n \\ \mathsf{proj}_2((w, n) :: ws) & = & n :: \mathsf{proj}_2(ws) \end{array}$$

We use $[\![\widehat{C}]\!]$ to compute a list of code sizes for $\widehat{C}$. Then

$\mathsf{proj}_2(ws) = [\![\widehat{C}]\!]$, where $\widehat{C}$ is some run-time labeled code and ws is the metric for $\widehat{C}$.

We define $[\![\widehat{C}]\!]$ as follows.

$$\begin{array}{rcl} [\![\mathbf{skip}^l]\!] & = & 0 \\ [\![c^l]\!] & = & 1 \\ [\![\langle C \rangle^l]\!] & = & 1 \\ [\![\widehat{C}_1; \widehat{C}_2]\!] & = & \begin{cases} [\![\widehat{C}_1]\!] \oplus |\widehat{C}_2| \oplus 1 & \text{if } \mathsf{minlabel}(\widehat{C}_1) = \mathsf{label}(\widehat{C}_2) \\ |\widehat{C}_2| :: ([\![\widehat{C}_1]\!] \oplus 1) & \text{if } \mathsf{minlabel}(\widehat{C}_1) > \mathsf{label}(\widehat{C}_2) \end{cases} \\ [\![\mathbf{if}^l(B)\ \widehat{C}_1\ \mathbf{else}\ \widehat{C}_2]\!] & = & max\{|\widehat{C}_1|, |\widehat{C}_2|\} + 1 \\ [\![\mathbf{while}^l(B)\ \widehat{C}]\!] & = & 1 \\ [\![\mathsf{while}^l(B)\ \widehat{C}]\!] & = & 0 :: 0 \end{array}$$

Here the static size of commands $|\widehat{C}|$ is defined as follows.

$$\begin{array}{rcl} |\mathbf{skip}^l| & = & 0 \\ |c^l| & = & 1 \\ |\langle C \rangle^l| & = & 1 \\ |\widehat{C}_1; \widehat{C}_2| & = & |\widehat{C}_1| + |\widehat{C}_2| + 1 \\ |\mathbf{if}^l(B)\ \widehat{C}_1\ \mathbf{else}\ \widehat{C}_2| & = & max\{|\widehat{C}_1|, |\widehat{C}_2|\} + 1 \\ |\mathbf{while}^l(B)\ \widehat{C}| & = & 1 \\ |\mathsf{while}^l(B)\ \widehat{C}| & = & 0 \end{array}$$

And $ns \oplus n$ is defined as follows:

$$ns \oplus n \stackrel{\text{def}}{=} \begin{cases} n_1 + n & \text{if } ns = n_1 \\ (n_1 + n) :: ns' & \text{if } ns = n_1 :: ns' \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Examples of ws Below we use a few simple examples to show how ws changes during an execution. The second dimension of the ws for the runtime labeled code $\widehat{C}$ coincides with the above definition $[\![\widehat{C}]\!]$.

		C	σ	ws
1		`while`^{1}`(i > 0) i--`^{2};	`i` $= 2$	$(0, 1)$
2	$\rightarrow$	`i--`^{2}; `while`^{1}`(i > 0) i--`^{2};	`i` $= 2$	$(0, 0) :: (1, 2)$
3	$\rightarrow$	`skip`^{2}; `while`^{1}`(i > 0) i--`^{2};	`i` $= 1$	$(0, 0) :: (1, 1)$
4	$\rightarrow$	`while`^{1}`(i > 0) i--`^{2};	`i` $= 1$	$(0, 0) :: (1, 0)$
5	$\rightarrow$	`i--`^{2}; `while`^{1}`(i > 0) i--`^{2};	`i` $= 1$	$(0, 0) :: (0, 2)$
6	$\rightarrow$	`skip`^{2}; `while`^{1}`(i > 0) i--`^{2};	`i` $= 0$	$(0, 0) :: (0, 1)$
7	$\rightarrow$	`while`^{1}`(i > 0) i--`^{2};	`i` $= 0$	$(0, 0) :: (0, 0)$
8	$\rightarrow$	`skip`^{1};	`i` $= 0$	$(0, 0)$

		C	σ	ws
1		`i:=2`[1]; `while`[1]`(i>0){ j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `}`	$\mathtt{i}=0,\mathtt{j}=0$	$(0,3)$
2	$\rightarrow$	`skip`[1]; `while`[1]`(i>0){ j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,2)$
3	$\rightarrow$	`while`[1]`(i>0){ j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,1)$
4	$\rightarrow$	`j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,0)::(1,6)$
5	$\rightarrow$	`skip`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=1$	$(0,0)::(1,5)$
6	$\rightarrow$	`while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=1$	$(0,0)::(1,4)$
7	$\rightarrow$	`j--`[3]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=1$	$(0,0)::(1,3)::(0,2)$
8	$\rightarrow$	`skip`[3]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,0)::(1,3)::(0,1)$
9	$\rightarrow$	`while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,0)::(1,3)::(0,0)$
10	$\rightarrow$	`skip`[2]; `i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,0)::(1,3)$
11	$\rightarrow$	`i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=2,\mathtt{j}=0$	$(0,0)::(1,2)$
12	$\rightarrow$	`skip`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(1,1)$
13	$\rightarrow$	`while`[1]`(i>0){ j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(1,0)$
14	$\rightarrow$	`j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(0,6)$
15	$\rightarrow$	`skip`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=1$	$(0,0)::(0,5)$
16	$\rightarrow$	`while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=1$	$(0,0)::(0,4)$
17	$\rightarrow$	`j--`[3]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=1$	$(0,0)::(0,3)::(0,2)$
18	$\rightarrow$	`skip`[3]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(0,3)::(0,1)$
19	$\rightarrow$	`while`[2]`(j>0){j--`[3]`;}; i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(0,3)::(0,0)$
20	$\rightarrow$	`skip`[2]; `i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(0,3)$
21	$\rightarrow$	`i--`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=1,\mathtt{j}=0$	$(0,0)::(0,2)$
22	$\rightarrow$	`skip`[2]; `while`[1]`(i>0){...}`	$\mathtt{i}=0,\mathtt{j}=0$	$(0,0)::(0,1)$
23	$\rightarrow$	`while`[1]`(i>0){ j:=1`[2]; `while`[2]`(j>0){j--`[3]`;}; i--`[2]; `}`	$\mathtt{i}=0,\mathtt{j}=0$	$(0,0)::(0,0)$
24	$\rightarrow$	`skip`[1]	$\mathtt{i}=0,\mathtt{j}=0$	$(0,0)$

The next example is a loop that uses the counter. It involves environment steps, denoted by R, and defined in Section 4.1. When the environment updates `x` (see line 7), we increase the number of tokens by 1, i.e., w at the outermost pair of the stack ws is increased from 0 to 1.

		C	σ	ws

1		`while`^{1}`(i > 0){`
		`  b:=false`^{2}`;`
		`  while`^{2}`(!b){ t:=x`^{3}`;b:=cas(&x,t,t+1)`^{3}`;if`^{3}`(b) i--`^{3}`; };`
		`}`
			$\mathtt{x}=5$
			$\mathtt{i}=1$
			$\mathtt{b}=\mathbf{false}$
			$\mathtt{t}=0$
				$(0,1)$

2	$\rightarrow$	`b:=false`^{2}`; while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	...	$(0,0)::(0,4)$
3	$\rightarrow$	`skip`^{2}`; while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	...	$(0,0)::(0,3)$
4	$\rightarrow$	`while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	...	$(0,0)::(0,2)$
5	$\rightarrow$	`t:=x`^{3}`;b:=cas(&x,t,t+1)`^{3}`;if`^{3}`(b) i--`^{3}`;` `while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	...	$(0,0)::(0,1)::(0,7)$
6	$\rightarrow$	`skip`^{3}`;b:=cas(&x,t,t+1)`^{3}`;if`^{3}`(b) i--`^{3}`;` `while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	$\mathtt{x}=5$ ... $\mathtt{t}=5$	$(0,0)::(0,1)::(0,6)$
7	R		$\mathtt{x}=8,\ldots$	$(0,0)::(0,1)::(1,6)$
8	$\rightarrow^*$	`while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	$\mathtt{x}=8$ $\mathtt{i}=1$ $\mathtt{b}=\mathbf{false}$ $\mathtt{t}=5$	$(0,0)::(0,1)::(1,0)$
9	$\rightarrow$	`t:=x`^{3}`;b:=cas(&x,t,t+1)`^{3}`;if`^{3}`(b) i--`^{3}`;` `while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	...	$(0,0)::(0,1)::(0,7)$
10	$\rightarrow^*$	`while`^{2}`(!b){...}; while`^{1}`(i > 0){...}`	$\mathtt{x}=8$ $\mathtt{i}=0$ $\mathtt{b}=\mathbf{true}$ $\mathtt{t}=8$	$(0,0)::(0,1)::(0,0)$
11	$\rightarrow$	`skip`^{2}`; while`^{1}`(i > 0){...}`	...	$(0,0)::(0,1)$
12	$\rightarrow$	`while`^{1}`(i > 0){...}`	...	$(0,0)::(0,0)$
13	$\rightarrow$	`skip`^{1}`;`	...	$(0,0)$

Note that in this section we assume that the outer loop and the inner loop each uses a "local" while-specific metric w. The intuition explained here actually shows how we prove the soundness of the WHILE-L rule. For the WHILE rule, we use a "global" while-specific metric, and hence the depth of ws could be just 1 and we do not need to push a new (w,n) pair whenever entering a loop. In this case, the second dimension of ws, i.e., the size of the code, will count in the runtime while command `while` $(B)\{C\}$ too. We show a simple example below, where the stack ws is always of depth 1.

		C	σ	ws
1		`while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=2$	$(2,1)$
2	$\rightarrow$	`i--`^{2}`; while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=2$	$(1,3)$
3	$\rightarrow$	`skip`^{2}`; while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=1$	$(1,2)$
4	$\rightarrow$	`while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=1$	$(1,0)$
5	$\rightarrow$	`i--`^{2}`; while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=1$	$(0,3)$
6	$\rightarrow$	`skip`^{2}`; while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=0$	$(0,2)$
7	$\rightarrow$	`while`^{1}`(i > 0) i--`^{2}`;`	$\mathtt{i}=0$	$(0,1)$
8	$\rightarrow$	`skip`^{1}`;`	$\mathtt{i}=0$	$(0,0)$

5.2.3 Unary Judgment Semantics

Definition 8. $R, G, I \models \{p\}C\{q\}$ iff
for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then $R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w;q} (\mathbb{D}, \Sigma)$.
Whenever $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$, then $(\sigma, \Sigma) \models I * \mathbf{true}$ and the following are true:

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:
   (a) either, there exist ws', w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{C}', \Sigma')$;
   (b) or, there exists ws' such that $ws' <_{\mathcal{H}} ws$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$;
2. for any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', ws', w', $\mathbb{C}'$ and Σ' such that $\sigma'' = \sigma' \uplus \sigma_F$, $(\mathbb{D}, \Sigma \uplus \Sigma_F) \xrightarrow{e}{}^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{C}', \Sigma')$;
3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$, then there exist ws' and w' such that $R, G, I \models (C, \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{D}, \Sigma')$;
4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$, then $R, G, I \models (C, \sigma', ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma')$;
5. if $C = \mathbf{skip}$, then for any Σ_F, if $\Sigma \perp \Sigma_F$, one of the following holds:
   (a) either, there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{C}', \Sigma') \models q$;
   (b) or, there exists w' such that $ws = (w', 0)$ and $(\sigma, w + w', \mathbb{D}, \Sigma) \models q$;
6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$ and $\Sigma \perp \Sigma_F$, then $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Definition 9 (SL Judgment Semantics).
$\models_{\text{SL}} [p]C[q]$ iff, for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, the following are true:

1. for any σ', if $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$, then $(\sigma', w, \mathbb{D}, \Sigma) \models q$;
2. $(C, \sigma) \not\longrightarrow^* \mathbf{abort}$;
3. $(C, \sigma) \not\longrightarrow^\omega \cdot$.

$\models_{\text{SL}} [P]\mathbb{C}[Q]$ iff, for any σ and Σ, if $(\sigma, \Sigma) \models P$, the following are true:

1. for any Σ', if $(\mathbb{C}, \Sigma) \longrightarrow^* (\boldsymbol{skip}, \Sigma')$, then $(\sigma, \Sigma') \models Q$;
2. $(\mathbb{C}, \Sigma) \not\longrightarrow^* \mathbf{abort}$;
3. $(\mathbb{C}, \Sigma) \not\longrightarrow^\omega \cdot$.

Definition 10 (Locality).
$\mathsf{Locality}(C)$ iff, for any σ_1 and σ_2, let $\sigma = \sigma_1 \uplus \sigma_2$, then the following hold:

1. (Safety monotonicity) If $(C, \sigma_1) \not\longrightarrow^* \mathbf{abort}$, then $(C, \sigma) \not\longrightarrow^* \mathbf{abort}$.
2. (Termination monotonicity) If $(C, \sigma_1) \not\longrightarrow^* \mathbf{abort}$ and $(C, \sigma_1) \not\longrightarrow^\omega \cdot$, then $(C, \sigma) \not\longrightarrow^\omega \cdot$.
3. (Frame property) For any n and σ', if $(C, \sigma_1) \not\longrightarrow^* \mathbf{abort}$ and $(C, \sigma) \longrightarrow^n (C', \sigma')$, then there exists σ_1' such that $\sigma' = \sigma_1' \uplus \sigma_2$ and $(C, \sigma_1) \longrightarrow^n (C', \sigma_1')$.

$\mathsf{Locality}(\mathbb{C})$ is defined similarly.

5.3 Soundness of Binary Rules

Lemma 11. If $R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}$, then $I \triangleright \{R, G\}$, $P \vee Q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(\{P, Q\}, R * \mathsf{Id})$.

Proof: By induction over the derivation of $R, G, I \vdash \{P\}C \preceq \mathbb{C}\{Q\}$, and by Lemma 27. For the stability, we need Lemmas 12, 13 and 14. □

Lemma 12. If $\mathsf{Sta}(p \wedge B, R * \mathsf{Id})$, $\mathsf{Sta}(p \wedge \neg B, R * \mathsf{Id})$ and $p \Rightarrow (B = B)$, then $\mathsf{Sta}(p, R * \mathsf{Id})$.

Lemma 13. If $\mathsf{Sta}(p, R * \mathsf{Id})$, $p \Rightarrow (B = B) * I$ and $I \triangleright R$, then $\mathsf{Sta}(p \wedge B, R * \mathsf{Id})$.

Lemma 14. If $\mathsf{Sta}(p_1, R_1 * \mathsf{Id})$, $\mathsf{Sta}(p_2, R_2 * \mathsf{Id})$, $I_1 \triangleright R_1$, $I_2 \triangleright R_2$, $p_1 \Rightarrow I_1 * \mathbf{true}$, $p_2 \Rightarrow I_2 * \mathbf{true}$, then $\mathsf{Sta}(p_1 * p_2, R_1 * R_2 * \mathsf{Id})$.

The B-PAR rule. We define $M_1 + M_2$ as a pair (M_1, M_2). The corresponding well-founded order satisfies the following:

$$(M_1 < M_2) \implies (M_1 + M_3 < M_2 + M_3) \tag{5.10}$$

$$(M_1 < M_2) \implies (M_3 + M_1 < M_3 + M_2) \tag{5.11}$$

Lemma 15 (Parallel Compositioinality). If

1. $R \vee G_2, G_1, I \models \{P_1 * P\}C_1 \preceq \mathbb{C}_1\{Q_1 * Q_1'\}$;
2. $R \vee G_1, G_2, I \models \{P_2 * P\}C_2 \preceq \mathbb{C}_2\{Q_2 * Q_2'\}$;
3. $P \vee Q_1' \vee Q_2' \Rightarrow I$; $I \triangleright \{R, G_1, G_2\}$; $\mathsf{Sta}(Q_1 * Q_1', (R \vee G_2) * \mathsf{Id})$; $\mathsf{Sta}(Q_2 * Q_2', (R \vee G_1) * \mathsf{Id})$;

then $R, G_1 \vee G_2, I \models \{P_1 * P_2 * P\}C_1 \,\|\, C_2 \preceq \mathbb{C}_1 \,|||\, \mathbb{C}_2\{Q_1 * Q_2 * (Q_1' \wedge Q_2')\}$.

Proof: We need to prove: for all σ and Σ, if $(\sigma, \Sigma) \models P_1 * P_2 * P$, then there exists M such that $R, G_1 \vee G_2, I \models (C_1 \,\|\, C_2, \sigma, M) \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma)$.

From $(\sigma, \Sigma) \models P_1 * P_2 * P$, we know there exist σ_1, σ_2, σ_r Σ_1, Σ_2 and Σ_r such that

$$(\sigma_1, \Sigma_1) \models P_1, \;\; (\sigma_2, \Sigma_2) \models P_2, \;\; (\sigma_r, \Sigma_r) \models P, \;\; \sigma = \sigma_1 \uplus \sigma_2 \uplus \sigma_r, \;\; \Sigma = \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r$$

From the premises, we know there exist M_1 and M_2 such that

$$R \vee G_2, G_1, I \models (C_1, \sigma_1 \uplus \sigma_r, M_1) \preceq_{Q_1 * Q_1'} (\mathbb{C}_1, \Sigma_1 \uplus \Sigma_r)$$
$$R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma_r, M_2) \preceq_{Q_2 * Q_2'} (\mathbb{C}_2, \Sigma_2 \uplus \Sigma_r)$$

By Lemma 16, we are done. □

Lemma 16. If

1. $R \vee G_2, G_1, I \models (C_1, \sigma_1 \uplus \sigma_r, M_1) \preceq_{Q_1 * Q_1'} (\mathbb{C}_1, \Sigma_1 \uplus \Sigma_r)$;
2. $R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma_r, M_2) \preceq_{Q_2 * Q_2'} (\mathbb{C}_2, \Sigma_2 \uplus \Sigma_r)$;
3. $(\sigma_r, \Sigma_r) \models I$; $Q_1' \vee Q_2' \Rightarrow I$; $I \triangleright \{R, G_1, G_2\}$; $\mathsf{Sta}(Q_1 * Q_1', (R \vee G_2) * \mathsf{Id})$; $\mathsf{Sta}(Q_2 * Q_2', (R \vee G_1) * \mathsf{Id})$;

then $R, G_1 \vee G_2, I \models (C_1 \,\|\, C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r, M_1 + M_2) \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r)$.

Proof: By co-induction. We know $(\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r) \models I * \mathbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(C_1 \,\|\, C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r \uplus \sigma_F) \longrightarrow (C', \sigma'')$, then one of the following three cases holds:

(a) $C' = C_1' \,\|\, C_2$ and $(C_1, \sigma_1 \uplus \sigma_2 \uplus \sigma_r \uplus \sigma_F) \longrightarrow (C_1', \sigma'')$:
from the premise 1, we know: there exists σ' such that

$$\sigma'' = \sigma' \uplus \sigma_2 \uplus \sigma_F \tag{5.12}$$

and one of the following holds:

i. there exist M_1', $\mathbb{C}_1'$ and Σ' such that

$$(\mathbb{C}_1, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}_1', \Sigma' \uplus \Sigma_2 \uplus \Sigma_F) \tag{5.13}$$

$$((\sigma_1 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_r), (\sigma', \Sigma'), \mathbf{true}) \models {G_1}^+ * \mathsf{True} \tag{5.14}$$

$$R \vee G_2, G_1, I \models (C_1', \sigma', M_1') \preceq_{Q_1 * Q_1'} (\mathbb{C}_1', \Sigma') \tag{5.15}$$

Below we prove 1(a) of Definition 2 holds.
From $I \triangleright G_1$, $(\sigma_r, \Sigma_r) \models I$ and (5.14), we know: there exist σ_1', Σ_1', σ_r' and Σ_r' such that

$$\sigma' = \sigma_1' \uplus \sigma_r', \quad \Sigma' = \Sigma_1' \uplus \Sigma_r', \quad (\sigma_r', \Sigma_r') \models I \tag{5.16}$$

$$((\sigma_r, \Sigma_r), (\sigma_r', \Sigma_r'), \mathbf{true}) \models {G_1}^+ \tag{5.17}$$

From (5.12) and (5.16), we know

$$\sigma'' = \sigma_1' \uplus \sigma_2 \uplus \sigma_r' \uplus \sigma_F \tag{5.18}$$

From (5.13) and (5.16), we know

$$(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}_1' \,|||\, \mathbb{C}_2, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r' \uplus \Sigma_F) \tag{5.19}$$

From (5.17), we know:

$$((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma_1' \uplus \sigma_2 \uplus \sigma_r', \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r'), \mathbf{true}) \models (G_1 \vee G_2)^+ * \mathsf{True} \tag{5.20}$$

and $((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r), (\sigma_2 \uplus \sigma_r', \Sigma_2 \uplus \Sigma_r'), \mathbf{true}) \models (G_1 \vee R)^+ * \mathsf{Id}$.
Then from the premise 2, we know: there exists M_2' such that

$$R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma_r', M_2') \preceq_{Q_2 * Q_2'} (\mathbb{C}_2, \Sigma_2 \uplus \Sigma_r') \tag{5.21}$$

From (5.15), (5.16), (5.21) and the co-induction hypothesis, we know:

$$R, G_1 \vee G_2, I \models (C_1' \,\|\, C_2, \sigma_1' \uplus \sigma_2 \uplus \sigma_r', M_1' + M_2') \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\mathbb{C}_1' \,|||\, \mathbb{C}_2, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r') \tag{5.22}$$

From (5.18), (5.19), (5.20) and (5.22), we are done.

ii. there exists M_1' such that

$$M_1' < M_1 \tag{5.23}$$

$$((\sigma_1 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_r), (\sigma', \Sigma_1 \uplus \Sigma_r), \mathbf{false}) \models {G_1}^+ * \mathsf{True} \tag{5.24}$$

$$R \vee G_2, G_1, I \models (C_1', \sigma', M_1') \preceq_{Q_1 * Q_1'} (\mathbb{C}_1, \Sigma_1 \uplus \Sigma_r) \tag{5.25}$$

Below we prove 1(b) of Definition 2 holds.
From $I \triangleright G_1$, $(\sigma_r, \Sigma_r) \models I$ and (5.24), we know: there exist σ_1' and σ_r' such that

$$\sigma' = \sigma_1' \uplus \sigma_r', \quad (\sigma_r', \Sigma_r) \models I \tag{5.26}$$

$$((\sigma_r, \Sigma_r), (\sigma_r', \Sigma_r), \mathbf{false}) \models {G_1}^+ \tag{5.27}$$

From (5.12) and (5.26), we know

$$\sigma'' = \sigma_1' \uplus \sigma_2 \uplus \sigma_r' \uplus \sigma_F \tag{5.28}$$

From (5.27), we know:

$$((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma_1' \uplus \sigma_2 \uplus \sigma_r', \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), \mathbf{false}) \models (G_1 \vee G_2)^+ * \mathsf{True} \quad (5.29)$$

and $((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r), (\sigma_2 \uplus \sigma_r', \Sigma_2 \uplus \Sigma_r), \mathbf{false}) \models (G_1 \vee R)^+ * \mathsf{Id}$.
Then from the premise 2, we know:

$$R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma_r', M_2) \preceq_{Q_2 * Q_2'} (\mathbb{C}, \Sigma_2 \uplus \Sigma_r) \quad (5.30)$$

From (5.25), (5.26), (5.30) and the co-induction hypothesis, we know:

$$R, G_1 \vee G_2, I \models (C_1' \parallel C_2, \sigma_1' \uplus \sigma_2 \uplus \sigma_r', M_1' + M_2) \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r) \quad (5.31)$$

From (5.23), we get:

$$M_1' + M_2 < M_1 + M_2 \quad (5.32)$$

From (5.28), (5.29), (5.31) and (5.32), we are done.

(b) $C' = C_1 \parallel C_2'$ and $(C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r \uplus \sigma_F) \longrightarrow (C_2', \sigma'')$: similar to the first case.

(c) $C' = \mathbf{skip}$, $C_1 = \mathbf{skip}$ and $C_2 = \mathbf{skip}$, thus we know

$$\sigma'' = \sigma_1 \uplus \sigma_2 \uplus \sigma_r \uplus \sigma_F \quad (5.33)$$

Below we prove 1(a) of Definition 2 holds.
From the premise 1, we know one of the following holds:

i. there exists Σ' such that

$$(\mathbb{C}_1, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\boldsymbol{skip}, \Sigma' \uplus \Sigma_2 \uplus \Sigma_F) \quad (5.34)$$

$$((\sigma_1 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_r, \Sigma'), \mathbf{true}) \models {G_1}^+ * \mathsf{True} \quad (5.35)$$

$$(\sigma_1 \uplus \sigma_r, \Sigma') \models Q_1 * Q_1' \quad (5.36)$$

From $I \triangleright G_1$, $(\sigma_r, \Sigma_r) \models I$ and (5.35), we know: there exist Σ_1' and Σ_r' such that

$$\Sigma' = \Sigma_1' \uplus \Sigma_r', \quad (\sigma_r, \Sigma_r') \models I \quad (5.37)$$

$$((\sigma_r, \Sigma_r), (\sigma_r, \Sigma_r'), \mathbf{true}) \models {G_1}^+ \quad (5.38)$$

Since $Q_1' \Rightarrow I$ and (5.36), we get:

$$(\sigma_1, \Sigma_1') \models Q_1, \quad (\sigma_r, \Sigma_r') \models Q_1' \quad (5.39)$$

From (5.34) and (5.37), we know

$$(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\boldsymbol{skip} \,|||\, \mathbb{C}_2, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r' \uplus \Sigma_F) \quad (5.40)$$

From (5.38), we know: $((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r), (\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r'), \mathbf{true}) \models (G_1 \vee R)^+ * \mathsf{Id}$.
Then from the premise 2, we know: there exists M_2' such that

$$R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma_r, M_2') \preceq_{Q_2 * Q_2'} (\mathbb{C}_2, \Sigma_2 \uplus \Sigma_r') \quad (5.41)$$

Since $C_2 = \mathbf{skip}$, we know one of the following holds:

A. there exists Σ'' such that

$$(\mathbb{C}_2, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r' \uplus \Sigma_F) \longrightarrow^+ (\textbf{\textit{skip}}, \Sigma'' \uplus \Sigma_1' \uplus \Sigma_F) \tag{5.42}$$

$$((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r'), (\sigma_2 \uplus \sigma_r, \Sigma''), \mathbf{true}) \models {G_2}^+ * \mathsf{True} \tag{5.43}$$

$$(\sigma_2 \uplus \sigma_r, \Sigma'') \models Q_2 * Q_2' \tag{5.44}$$

From $I \triangleright G_2$, $(\sigma_r, \Sigma_r') \models I$ and (5.43), we know: there exist Σ_2' and Σ_r'' such that

$$\Sigma'' = \Sigma_2' \uplus \Sigma_r'', \quad (\sigma_r, \Sigma_r'') \models I \tag{5.45}$$

$$((\sigma_r, \Sigma_r'), (\sigma_r, \Sigma_r''), \mathbf{true}) \models {G_2}^+ \tag{5.46}$$

Since $Q_2' \Rightarrow I$ and (5.44), we get:

$$(\sigma_2, \Sigma_2') \models Q_2, \quad (\sigma_r, \Sigma_r'') \models Q_2' \tag{5.47}$$

From (5.40) and (5.42), we know

$$(\mathbb{C}_1 ||| \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\textbf{\textit{skip}}, \Sigma_1' \uplus \Sigma_2' \uplus \Sigma_r'' \uplus \Sigma_F) \tag{5.48}$$

From (5.38) and (5.46), we know:

$$((\sigma_r, \Sigma_r), (\sigma_r, \Sigma_r''), \mathbf{true}) \models (G_1 \vee G_2)^+ \tag{5.49}$$

Thus we get:

$$((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1' \uplus \Sigma_2' \uplus \Sigma_r''), \mathbf{true}) \models (G_1 \vee G_2)^+ * \mathsf{True} \tag{5.50}$$

From (5.46), we get: $((\sigma_r, \Sigma_r'), (\sigma_r, \Sigma_r''), \mathbf{true}) \models (R \vee G_2)^+$. Since $(\sigma_1, \Sigma_1') \models Q_1$, $(\sigma_r, \Sigma_r') \models Q_1'$, $\mathsf{Sta}(Q_1 * Q_1', (R \vee G_2) * \mathsf{Id})$, $I \triangleright (R \vee G_2)$ and $Q_1' \Rightarrow I$, we know:

$$(\sigma_r, \Sigma_r'') \models Q_1' \tag{5.51}$$

From $(\sigma_1, \Sigma_1') \models Q_1$ and (5.47), we get:

$$(\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1' \uplus \Sigma_2' \uplus \Sigma_r'') \models Q_1 * Q_2 * (Q_1' \wedge Q_2') \tag{5.52}$$

By the B-SKIP and B-FRAME rules, we get: there exists M' such that

$$R, G_1 \vee G_2, I \models (\mathbf{skip}, \sigma_1 \uplus \sigma_2 \uplus \sigma_r, M') \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\textbf{\textit{skip}}, \Sigma_1' \uplus \Sigma_2' \uplus \Sigma_r'') \tag{5.53}$$

From (5.48), (5.50) and (5.53), we are done.

B. $\mathbb{C}_2 = \textbf{\textit{skip}}$ and $(\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r') \models Q_2 * Q_2'$.
From $Q_2' \Rightarrow I$ and $(\sigma_r, \Sigma_r') \models I$, we know:

$$(\sigma_2, \Sigma_2) \models Q_2, \quad (\sigma_r, \Sigma_r') \models Q_2' \tag{5.54}$$

From (5.40), we know

$$(\mathbb{C}_1 ||| \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\textbf{\textit{skip}}, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r' \uplus \Sigma_F) \tag{5.55}$$

From (5.38), we know:

$$((\sigma_r, \Sigma_r), (\sigma_r, \Sigma_r'), \mathbf{true}) \models (G_1 \vee G_2)^+ \tag{5.56}$$

Thus we get:

$$((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r'), \mathbf{true}) \models (G_1 \vee G_2)^+ * \mathsf{True} \tag{5.57}$$

From (5.39) and (5.54), we get:

$$(\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r') \models Q_1 * Q_2 * (Q_1' \wedge Q_2') \tag{5.58}$$

By the B-SKIP and B-FRAME rules, we get: there exists M' such that

$$R, G_1 \vee G_2, I \models (\mathbf{skip}, \sigma_1 \uplus \sigma_2 \uplus \sigma_r, M') \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\boldsymbol{skip}, \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_r') \tag{5.59}$$

From (5.55), (5.57) and (5.59), we are done.

ii. $\mathbb{C}_1 = \boldsymbol{skip}$ and $(\sigma_1 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_r) \models Q_1 * Q_1'$.
From $Q_1' \Rightarrow I$ and $(\sigma_r, \Sigma_r) \models I$, we know:

$$(\sigma_1, \Sigma_1) \models Q_1\,, \quad (\sigma_r, \Sigma_r) \models Q_1' \tag{5.60}$$

From the premise 2, we know one of the following holds:

A. there exists Σ' such that

$$(\mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\boldsymbol{skip}, \Sigma' \uplus \Sigma_1 \uplus \Sigma_F) \tag{5.61}$$

$$((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r), (\sigma_2 \uplus \sigma_r, \Sigma'), \mathbf{true}) \models {G_2}^+ * \mathsf{True} \tag{5.62}$$

$$(\sigma_2 \uplus \sigma_r, \Sigma') \models Q_2 * Q_2' \tag{5.63}$$

From $I \triangleright G_2$, $(\sigma_r, \Sigma_r) \models I$ and (5.62), we know: there exist Σ_2' and Σ_r' such that

$$\Sigma' = \Sigma_2' \uplus \Sigma_r'\,, \quad (\sigma_r, \Sigma_r') \models I \tag{5.64}$$

$$((\sigma_r, \Sigma_r), (\sigma_r, \Sigma_r'), \mathbf{true}) \models {G_2}^+ \tag{5.65}$$

Since $Q_2' \Rightarrow I$ and (5.63), we get:

$$(\sigma_2, \Sigma_2') \models Q_2\,, \quad (\sigma_r, \Sigma_r') \models Q_2' \tag{5.66}$$

From (5.61), we know

$$(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\boldsymbol{skip}, \Sigma_1 \uplus \Sigma_2' \uplus \Sigma_r' \uplus \Sigma_F) \tag{5.67}$$

From (5.65), we know:

$$((\sigma_r, \Sigma_r), (\sigma_r, \Sigma_r'), \mathbf{true}) \models (G_1 \vee G_2)^+ \tag{5.68}$$

Thus we get:

$$((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2' \uplus \Sigma_r'), \mathbf{true}) \models (G_1 \vee G_2)^+ * \mathsf{True} \tag{5.69}$$

From (5.65), we get: $((\sigma_r, \Sigma_r), (\sigma_r, \Sigma_r'), \mathbf{true}) \models (R \vee G_2)^+$. Since $(\sigma_1, \Sigma_1) \models Q_1$, $(\sigma_r, \Sigma_r) \models Q_1'$, $\mathsf{Sta}(Q_1 * Q_1', (R \vee G_2) * \mathsf{Id})$, $I \triangleright (R \vee G_2)$ and $Q_1' \Rightarrow I$, we know:

$$(\sigma_r, \Sigma_r') \models Q_1' \tag{5.70}$$

From $(\sigma_1, \Sigma_1) \models Q_1$ and (5.66), we get:

$$(\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2' \uplus \Sigma_r') \models Q_1 * Q_2 * (Q_1' \wedge Q_2') \tag{5.71}$$

By the B-SKIP and B-FRAME rules, we get: there exists M' such that

$$R, G_1 \vee G_2, I \models (\mathbf{skip}, \sigma_1 \uplus \sigma_2 \uplus \sigma_r, M') \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\boldsymbol{skip}, \Sigma_1 \uplus \Sigma_2' \uplus \Sigma_r') \tag{5.72}$$

From (5.67), (5.69) and (5.72), we are done.

B. $\mathbb{C}_2 = \textbf{\textit{skip}}$ and $(\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r) \models Q_2 * Q_2'$.
From $Q_2' \Rightarrow I$ and $(\sigma_r, \Sigma_r) \models I$, we know:

$$(\sigma_2, \Sigma_2) \models Q_2\,, \quad (\sigma_r, \Sigma_r) \models Q_2' \tag{5.73}$$

We know

$$(\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ (\textbf{\textit{skip}}, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \tag{5.74}$$

Also we have:

$$((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), \mathbf{true}) \models (G_1 \vee G_2)^+ * \mathsf{True} \tag{5.75}$$

From (5.60) and (5.73), we get:

$$(\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r) \models Q_1 * Q_2 * (Q_1' \wedge Q_2') \tag{5.76}$$

By the B-SKIP and B-FRAME rules, we get: there exists M' such that

$$R, G_1 \vee G_2, I \models (\mathbf{skip}, \sigma_1 \uplus \sigma_2 \uplus \sigma_r, M') \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\textbf{\textit{skip}}, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r) \tag{5.77}$$

From (5.74), (5.75) and (5.77), we are done.

2. for any σ_F, Σ_F, e, C' and σ'', if $(C_1 \,\|\, C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$, the proof is similar to the first case.

3. for any σ' and Σ', if $((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,
from $I \triangleright R$ and $(\sigma_r, \Sigma_r) \models I$, we know: there exist σ_r' and Σ_r' such that

$$\sigma' = \sigma_1 \uplus \sigma_2 \uplus \sigma_r'\,, \quad \Sigma' = \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r'\,, \quad (\sigma_r', \Sigma_r') \models I \tag{5.78}$$

$$((\sigma_r, \Sigma_r), (\sigma_r', \Sigma_r'), \mathbf{true}) \models R^+ \tag{5.79}$$

Thus we get:

$$((\sigma_1 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_r', \Sigma_1 \uplus \Sigma_r'), \mathbf{true}) \models (R \vee G_2)^+ * \mathsf{Id} \tag{5.80}$$

$$((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r), (\sigma_2 \uplus \sigma_r', \Sigma_2 \uplus \Sigma_r'), \mathbf{true}) \models (R \vee G_1)^+ * \mathsf{Id} \tag{5.81}$$

From the premises, we know: there exist M_1' and M_2' such that

$$R \vee G_2, G_1, I \models (C_1, \sigma_1 \uplus \sigma_r', M_1') \preceq_{Q_1 * Q_1'} (\mathbb{C}_1, \Sigma_1 \uplus \Sigma_r') \tag{5.82}$$

$$R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma_r', M_2') \preceq_{Q_2 * Q_2'} (\mathbb{C}_2, \Sigma_2 \uplus \Sigma_r') \tag{5.83}$$

By the co-induction hypothesis, we get:

$$R, G_1 \vee G_2, I \models (C_1 \,\|\, C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r', M_1' + M_2') \preceq_{Q_1 * Q_2 * (Q_1' \wedge Q_2')} (\mathbb{C}_1 \,|||\, \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r') \tag{5.84}$$

4. for any σ' and Σ', if $((\sigma_1 \uplus \sigma_2 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$,
from $I \triangleright R$ and $(\sigma_r, \Sigma_r) \models I$, we know: there exist σ_r' and Σ_r' such that

$$\sigma' = \sigma_1 \uplus \sigma_2 \uplus \sigma_r'\,, \quad \Sigma' = \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r'\,, \quad (\sigma_r', \Sigma_r') \models I \tag{5.85}$$

$$((\sigma_r, \Sigma_r), (\sigma_r', \Sigma_r'), \mathbf{false}) \models R^+ \tag{5.86}$$

Thus we get:

$$((\sigma_1 \uplus \sigma_r, \Sigma_1 \uplus \Sigma_r), (\sigma_1 \uplus \sigma_r', \Sigma_1 \uplus \Sigma_r'), \mathbf{false}) \models (R \vee G_2)^+ * \mathsf{Id} \tag{5.87}$$

$$((\sigma_2 \uplus \sigma_r, \Sigma_2 \uplus \Sigma_r), (\sigma_2 \uplus \sigma_r', \Sigma_2 \uplus \Sigma_r'), \mathbf{false}) \models (R \vee G_1)^+ * \mathsf{Id} \tag{5.88}$$

From the premises, we know:

$$R \vee G_2, G_1, I \models (C_1, \sigma_1 \uplus \sigma'_r, M_1) \preceq_{Q_1 * Q'_1} (\mathbb{C}_1, \Sigma_1 \uplus \Sigma'_r) \tag{5.89}$$

$$R \vee G_1, G_2, I \models (C_2, \sigma_2 \uplus \sigma'_r, M_2) \preceq_{Q_2 * Q'_2} (\mathbb{C}_2, \Sigma_2 \uplus \Sigma'_r) \tag{5.90}$$

By the co-induction hypothesis, we get:

$$R, G_1 \vee G_2, I \models (C_1 \parallel C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma'_r, M_1 + M_2) \preceq_{Q_1 * Q_2 * (Q'_1 \wedge Q'_2)} (\mathbb{C}_1 \parallel\!\parallel \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma'_r) \tag{5.91}$$

5. for any σ_F and Σ_F, if $(C_1 \parallel C_2, \sigma_1 \uplus \sigma_2 \uplus \sigma_r \uplus \sigma_F) \longrightarrow \textbf{abort}$, by the operational semantics and the premises, we know $(\mathbb{C}_1 \parallel\!\parallel \mathbb{C}_2, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_r \uplus \Sigma_F) \longrightarrow^+ \textbf{abort}$.

Thus we are done. □

The U2B rule.

Lemma 17 (U2B). If $R, G, I \models \{P \wedge \mathsf{arem}(\mathbb{C})\} C \{Q \wedge \mathsf{arem}(\boldsymbol{skip})\}$, then $R, G, I \models \{P\} C \preceq \mathbb{C} \{Q\}$.

Proof: We need to prove: for all σ and Σ, if $(\sigma, \Sigma) \models P$, then there exists M such that $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$.

From $(\sigma, \Sigma) \models P$, we know: $(\sigma, 0, \mathbb{C}, \Sigma) \models P \wedge \mathsf{arem}(\mathbb{C})$.

From the premise, we know: $R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);0;Q \wedge \mathsf{arem}(\boldsymbol{skip})} (\mathbb{C}, \Sigma)$.

By Lemma 18, we are done. □

Lemma 18. If $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;Q \wedge \mathsf{arem}(\boldsymbol{skip})} (\mathbb{C}, \Sigma)$, then $R, G, I \models (C, \sigma, (ws, \mathcal{H})) \preceq_Q (\mathbb{C}, \Sigma)$.

Proof: By co-induction. From the premise, we know $(\sigma, \Sigma) \models I * \mathbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \perp \Sigma_F$,

   from the premise, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

   (a) there exist ws', w', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w';Q \wedge \mathsf{arem}(\boldsymbol{skip})} (\mathbb{C}', \Sigma')$.
   By the co-induction hypothesis, we know: $R, G, I \models (C', \sigma', (ws', \mathcal{H})) \preceq_Q (\mathbb{C}', \Sigma')$.

   (b) there exists ws' such that $ws' <_{\mathcal{H}} ws$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w;Q \wedge \mathsf{arem}(\boldsymbol{skip})} (\mathbb{C}, \Sigma)$.
   By the co-induction hypothesis, we know: $R, G, I \models (C', \sigma', (ws', \mathcal{H})) \preceq_Q (\mathbb{C}, \Sigma)$.
   By the instantiation of the abstract metric, we know: $(ws', \mathcal{H}) < (ws, \mathcal{H})$.

2. for any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$, the proof is similar to the previous case.

3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,

   from the premise, we know: there exist ws' and w' such that $R, G, I \models (C, \sigma', ws') \preceq_{\mathcal{H};w';Q \wedge \mathsf{arem}(\boldsymbol{skip})} (\mathbb{C}, \Sigma')$.

   By the co-induction hypothesis, we know: $R, G, I \models (C, \sigma', (ws', \mathcal{H})) \preceq_Q (\mathbb{C}, \Sigma')$.

4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$,

   from the premise, we know: $R, G, I \models (C, \sigma', ws) \preceq_{\mathcal{H};w;Q \wedge \mathsf{arem}(\boldsymbol{skip})} (\mathbb{C}, \Sigma')$.

   By the co-induction hypothesis, we know: $R, G, I \models (C, \sigma', (ws, \mathcal{H})) \preceq_Q (\mathbb{C}, \Sigma')$.

5. if $C = \mathbf{skip}$, then for any Σ_F, from the premise, we know one of the following holds:

   (a) there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{C}', \Sigma') \models Q \wedge \mathsf{arem}(\boldsymbol{skip})$.
   Thus we know $\mathbb{C}' = \boldsymbol{skip}$ and $(\sigma, \Sigma') \models Q$.

   (b) there exists w' such that $ws = (w', 0)$ and $(\sigma, w + w', \mathbb{C}, \Sigma) \models Q \wedge \mathsf{arem}(\boldsymbol{skip})$.
   Thus we know $\mathbb{C} = \boldsymbol{skip}$ and $(\sigma, \Sigma) \models Q$.

6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$, from the premise, we know $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

The TRANS rule. We define $M_2 \circ M_1$ as a pair (M_2, M_1) and the corresponding well-founded order as the lexical order. That is, the following hold:

$$(M_2 < M_2') \implies (M_2 \circ M_1 < M_2' \circ M_1') \tag{5.92}$$

$$(M_1 < M_1') \implies (M_2 \circ M_1 < M_2 \circ M_1') \tag{5.93}$$

Lemma 19 (TRANS). If

1. $R_1, G_1, I_1 \vdash \{P_1\} C \preceq \mathrm{C_M} \{Q_1\}$;
2. $R_2, G_2, I_2 \vdash \{P_2\} \mathrm{C_M} \preceq \mathbb{C} \{Q_2\}$;
3. $\mathsf{MPrecise}(I_1, I_2)$; $I_1 \triangleright \{R_1, G_1\}$; $I_2 \triangleright \{R_2, G_2\}$;
4. $((G_1)^{I_1} \mathbin{\hat{\S}} (G_2)^{I_2}) \;\Rightarrow\; (G_1 \mathbin{\hat{\S}} G_2)^{I_1 \S I_2}$; $(R_1 \mathbin{\check{\S}} R_2)^{I_1 \S I_2} \;\Rightarrow\; ((R_1)^{I_1} \mathbin{\check{\S}} (R_2)^{I_2})$;

then $(R_1 \mathbin{\check{\S}} R_2), (G_1 \mathbin{\hat{\S}} G_2), (I_1 \S I_2) \vdash \{P_1 \S P_2\} C \preceq \mathbb{C} \{Q_1 \S Q_2\}$.

Proof: For all σ and Σ, if $(\sigma, \Sigma) \models P_1 \S P_2$, we know there exists θ such that $(\sigma, \theta) \models P_1$ and $(\theta, \Sigma) \models P_2$. From the premise, we know:

1. there exists M_1 such that $R_1, G_1, I_1 \models (C, \sigma, M_1) \preceq_{Q_1} (\mathrm{C_M}, \theta)$.
2. there exists M_2 such that $R_2, G_2, I_2 \models (\mathrm{C_M}, \theta, M_2) \preceq_{Q_2} (\mathbb{C}, \Sigma)$.

By Lemma 20, we know $(R_1 \mathbin{\check{\S}} R_2), (G_1 \mathbin{\hat{\S}} G_2), (I_1 \S I_2) \models (C, \sigma, (M_2 \circ M_1)) \preceq_{Q_1 \S Q_2} (\mathbb{C}, \Sigma)$. Thus we are done. □

Lemma 20. If

1. $R_1, G_1, I_1 \models (C, \sigma, M_1) \preceq_{Q_1} (\mathrm{C_M}, \theta)$;
2. $R_2, G_2, I_2 \models (\mathrm{C_M}, \theta, M_2) \preceq_{Q_2} (\mathbb{C}, \Sigma)$;
3. $\mathsf{MPrecise}(I_1, I_2)$; $I_1 \triangleright \{R_1, G_1\}$; $I_2 \triangleright \{R_2, G_2\}$;
4. $((G_1)^+ \mathbin{\hat{\S}} (G_2)^+) \;\Rightarrow\; (G_1 \mathbin{\hat{\S}} G_2)^+$; $(R_1 \mathbin{\check{\S}} R_2)^+ \;\Rightarrow\; ((R_1)^+ \mathbin{\check{\S}} (R_2)^+)$;

then $(R_1 \mathbin{\check{\S}} R_2), (G_1 \mathbin{\hat{\S}} G_2), (I_1 \S I_2) \models (C, \sigma, (M_2 \circ M_1)) \preceq_{Q_1 \S Q_2} (\mathbb{C}, \Sigma)$.

Proof: By co-induction. By the premises, we know $(\sigma, \theta) \models I_1 * \mathbf{true}$ and $(\theta, \Sigma) \models I_2 * \mathbf{true}$. Since $\mathsf{MPrecise}(I_1, I_2)$, we know $(\sigma, \Sigma) \models (I_1 \S I_2) * \mathbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$, then by the premise 1, we know:
   there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and for any θ_F, one of the following holds:
   (a) either, there exist M_1', $\mathrm{C_M'}$ and θ' such that $(\mathrm{C_M}, \theta \uplus \theta_F) \longrightarrow^+ (\mathrm{C_M'}, \theta' \uplus \theta_F)$, $((\sigma, \theta), (\sigma', \theta'), \mathbf{true}) \models (G_1)^+ * \mathsf{True}$ and $R_1, G_1, I_1 \models (C', \sigma', M_1') \preceq_{Q_1} (\mathrm{C_M'}, \theta')$.
   By the premise 2 and Lemma 21, we know: one of the following holds:
      i. either, there exist M_2', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\theta, \Sigma), (\theta', \Sigma'), \mathbf{true}) \models (G_2)^+ * \mathsf{True}$ and $R_2, G_2, I_2 \models (\mathrm{C_M'}, \theta', M_2') \preceq_{Q_2} (\mathbb{C}', \Sigma')$.
      Thus we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models ((G_1)^+ * \mathsf{True}) \mathbin{\hat{\S}} ((G_2)^+ * \mathsf{True}) \tag{5.94}$$

      Since $I_1 \triangleright G_1$ and $I_2 \triangleright G_2$, we know $I_1 \triangleright (G_1)^+$ and $I_2 \triangleright (G_2)^+$. Since $\mathsf{MPrecise}(I_1, I_2)$, by Lemma 25, we know

$$((G_1)^+ * \mathsf{True}) \mathbin{\hat{\S}} ((G_2)^+ * \mathsf{True}) \;\Rightarrow\; ((G_1)^+ \mathbin{\hat{\S}} (G_2)^+) * \mathsf{True} \tag{5.95}$$

Thus we get:

$$((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models (G_1 \hat{\S} G_2)^+ * \mathsf{True} \tag{5.96}$$

Besides, by the co-induction hypothesis, we get:

$$(R_1 \tilde{\S} R_2),(G_1 \hat{\S} G_2),(I_1 \S I_2) \models (C',\sigma',(M_2' \circ M_1')) \preceq_{Q_1 \S Q_2} (\mathbb{C}',\Sigma') \tag{5.97}$$

ii. or, there exists M_2' such that $M_2' < M_2$,
$((\theta,\Sigma),(\theta',\Sigma),\mathbf{false}) \models (G_2)^+ * \mathsf{True}$ and $R_2,G_2,I_2 \models (\mathrm{C}_\mathrm{M}',\theta',M_2') \preceq_{Q_2} (\mathbb{C},\Sigma)$.
Thus we know

$$((\sigma,\Sigma),(\sigma',\Sigma),\mathbf{false}) \models ((G_1)^+ * \mathsf{True}) \hat{\S} ((G_2)^+ * \mathsf{True}) \tag{5.98}$$

Thus we get:

$$((\sigma,\Sigma),(\sigma',\Sigma),\mathbf{false}) \models (G_1 \hat{\S} G_2)^+ * \mathsf{True} \tag{5.99}$$

Besides, by the co-induction hypothesis, we get:

$$(R_1 \tilde{\S} R_2),(G_1 \hat{\S} G_2),(I_1 \S I_2) \models (C',\sigma',(M_2' \circ M_1')) \preceq_{Q_1 \S Q_2} (\mathbb{C},\Sigma) \tag{5.100}$$

Moreover, we know

$$(M_2' \circ M_1') < (M_2 \circ M_1) \tag{5.101}$$

(b) or, there exists M_1' such that $M_1' < M_1$,
$((\sigma,\theta),(\sigma',\theta),\mathbf{false}) \models (G_1)^+ * \mathsf{True}$ and $R_1,G_1,I_1 \models (C',\sigma',M_1') \preceq_{Q_1} (\mathrm{C}_\mathrm{M},\theta)$.
Since $(\theta,\Sigma) \models I_2 * \mathbf{true}$, we know $((\theta,\Sigma),(\theta,\Sigma),\mathbf{false}) \models (G_2)^+ * \mathsf{True}$. Thus

$$((\sigma,\Sigma),(\sigma',\Sigma),\mathbf{false}) \models ((G_1)^+ * \mathsf{True}) \hat{\S} ((G_2)^+ * \mathsf{True}) \tag{5.102}$$

Thus we get:

$$((\sigma,\Sigma),(\sigma',\Sigma),\mathbf{false}) \models (G_1 \hat{\S} G_2)^+ * \mathsf{True} \tag{5.103}$$

Besides, by the co-induction hypothesis, we get:

$$(R_1 \tilde{\S} R_2),(G_1 \hat{\S} G_2),(I_1 \S I_2) \models (C',\sigma',(M_2 \circ M_1')) \preceq_{Q_1 \S Q_2} (\mathbb{C},\Sigma) \tag{5.104}$$

Moreover, we know

$$(M_2 \circ M_1') < (M_2 \circ M_1) \tag{5.105}$$

2. for any σ_F, Σ_F, e, C' and σ'', if $(C,\sigma \uplus \sigma_F) \xrightarrow{e} (C',\sigma'')$, then by the premise 1, we know: for any θ_F, there exist σ', M_1', C_M' and θ' such that $\sigma'' = \sigma' \uplus \sigma_F$, $(\mathrm{C}_\mathrm{M},\theta \uplus \theta_F) \xrightarrow{e}{}^+ (\mathrm{C}_\mathrm{M}',\theta' \uplus \theta_F)$, $((\sigma,\theta),(\sigma',\theta'),\mathbf{true}) \models (G_1)^+ * \mathsf{True}$ and $R_1,G_1,I_1 \models (C',\sigma',M_1') \preceq_{Q_1} (\mathrm{C}_\mathrm{M}',\theta')$.

By the premise 2 and Lemma 22, we know:

there exist M_2', $\mathbb{C}'$ and Σ' such that $(\mathbb{C},\Sigma \uplus \Sigma_F) \xrightarrow{e}{}^+ (\mathbb{C}',\Sigma' \uplus \Sigma_F)$, $((\theta,\Sigma),(\theta',\Sigma'),\mathbf{true}) \models (G_2)^+ * \mathsf{True}$ and $R_2,G_2,I_2 \models (\mathrm{C}_\mathrm{M}',\theta',M_2') \preceq_{Q_2} (\mathbb{C}',\Sigma')$.

Thus we know

$$((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models ((G_1)^+ * \mathsf{True}) \hat{\S} ((G_2)^+ * \mathsf{True}) \tag{5.106}$$

Thus we get:

$$((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models (G_1 \hat{\S} G_2)^+ * \mathsf{True} \tag{5.107}$$

Besides, by the co-induction hypothesis, we get:

$$(R_1 \tilde{\S} R_2),(G_1 \hat{\S} G_2),(I_1 \S I_2) \models (C',\sigma',(M_2' \circ M_1')) \preceq_{Q_1 \S Q_2} (\mathbb{C}',\Sigma') \tag{5.108}$$

3. for any σ' and Σ', if $((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models (R_1 \mathbin{\tilde{;}} R_2)^+ * \mathsf{Id}$, then we know

$$((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models ((R_1)^+ \mathbin{\tilde{;}} (R_2)^+) * \mathsf{Id} \tag{5.109}$$

By Lemma 26, we know

$$((R_1)^+ \mathbin{\tilde{;}} (R_2)^+) * \mathsf{Id} \;\Rightarrow\; ((R_1)^+ * \mathsf{Id}) \mathbin{\tilde{;}} ((R_2)^+ * \mathsf{Id}) \tag{5.110}$$

Thus we get: there exist θ, θ', b_1 and b_2 such that $b = b_1 \vee b_2$,

$$((\sigma,\theta),(\sigma',\theta'),b_1) \models (R_1)^+ * \mathsf{Id} \;\text{ and }\; ((\theta,\Sigma),(\theta',\Sigma'),b_2) \models (R_2)^+ * \mathsf{Id} \tag{5.111}$$

From the premises, we know: there exist M_1' and M_2' such that

(a) $R_1, G_1, I_1 \models (C,\sigma',M_1') \preceq_{Q_1} (\mathrm{C_M},\theta')$;
(b) $R_2, G_2, I_2 \models (\mathrm{C_M},\theta',M_2') \preceq_{Q_2} (\mathbb{C},\Sigma')$.

By the co-induction hypothesis, we get:

$$(R_1 \mathbin{\tilde{;}} R_2),(G_1 \mathbin{\hat{;}} G_2),(I_1 \mathbin{;} I_2) \models (C,\sigma',(M_2' \circ M_1')) \preceq_{Q_1 ; Q_2} (\mathbb{C},\Sigma') \tag{5.112}$$

4. for any σ' and Σ', if $((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{false}) \models (R_1 \mathbin{\tilde{;}} R_2)^+ * \mathsf{Id}$, then we know

$$((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{false}) \models ((R_1)^+ * \mathsf{Id}) \mathbin{\tilde{;}} ((R_2)^+ * \mathsf{Id}) \tag{5.113}$$

Thus we get: there exist θ and θ' such that

$$((\sigma,\theta),(\sigma',\theta'),\mathbf{false}) \models (R_1)^+ * \mathsf{Id} \;\text{ and }\; ((\theta,\Sigma),(\theta',\Sigma'),\mathbf{false}) \models (R_2)^+ * \mathsf{Id} \tag{5.114}$$

From the premises, we know:

(a) $R_1, G_1, I_1 \models (C,\sigma',M_1) \preceq_{Q_1} (\mathrm{C_M},\theta')$;
(b) $R_2, G_2, I_2 \models (\mathrm{C_M},\theta',M_2) \preceq_{Q_2} (\mathbb{C},\Sigma')$.

By the co-induction hypothesis, we get:

$$(R_1 \mathbin{\tilde{;}} R_2),(G_1 \mathbin{\hat{;}} G_2),(I_1 \mathbin{;} I_2) \models (C,\sigma',(M_2 \circ M_1)) \preceq_{Q_1 ; Q_2} (\mathbb{C},\Sigma') \tag{5.115}$$

5. if $C = \mathbf{skip}$, then by the premise 1, we know: for any θ_F, one of the following holds:

   (a) either, there exists θ' such that $(\mathrm{C_M},\theta \uplus \theta_F) \longrightarrow^+ (\mathbf{skip},\theta' \uplus \theta_F)$, $((\sigma,\theta),(\sigma,\theta'),\mathbf{true}) \models (G_1)^+ * \mathsf{True}$ and $(\sigma,\theta') \models Q_1$.
   By the premise 2 and Lemma 23, we know: for any Σ_F, one of the following holds:

      i. there exists Σ' such that $(\mathbb{C},\Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbf{skip},\Sigma' \uplus \Sigma_F)$, $((\theta,\Sigma),(\theta',\Sigma'),\mathbf{true}) \models (G_2)^+ * \mathsf{True}$ and $(\theta',\Sigma') \models Q_2$.
      Thus we know

$$((\sigma,\Sigma),(\sigma,\Sigma'),\mathbf{true}) \models ((G_1)^+ * \mathsf{True}) \mathbin{\hat{;}} ((G_2)^+ * \mathsf{True}) \tag{5.116}$$

      Thus we get:

$$((\sigma,\Sigma),(\sigma,\Sigma'),\mathbf{true}) \models (G_1 \mathbin{\hat{;}} G_2)^+ * \mathsf{True} \tag{5.117}$$

      Besides, we get:

$$(\sigma,\Sigma') \models (Q_1 \mathbin{;} Q_2) \tag{5.118}$$

ii. or, $\mathbb{C} = \textbf{skip}$, $((\theta, \Sigma), (\theta', \Sigma), \textbf{false}) \models (G_2)^+ * \mathsf{True}$ and $(\theta', \Sigma) \models Q_2$.
We get:

$$(\sigma, \Sigma) \models (Q_1 \fatsemi Q_2) \tag{5.119}$$

(b) or, $\mathrm{C_M} = \textbf{skip}$ and $(\sigma, \theta) \models Q_1$.

By the premise 2, we know one of the following holds:

i. there exists Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\textbf{skip}, \Sigma' \uplus \Sigma_F)$, $((\theta, \Sigma), (\theta, \Sigma'), \textbf{true}) \models (G_2)^+ * \mathsf{True}$ and $(\theta, \Sigma') \models Q_2$.
Since $(\sigma, \theta) \models I_1 * \textbf{true}$, we know: $((\sigma, \theta), (\sigma, \theta), \textbf{true}) \models (G_1)^+ * \mathsf{True}$.
Thus we know

$$((\sigma, \Sigma), (\sigma, \Sigma'), \textbf{true}) \models ((G_1)^+ * \mathsf{True}) \hat{\fatsemi} ((G_2)^+ * \mathsf{True}) \tag{5.120}$$

Thus we get:

$$((\sigma, \Sigma), (\sigma, \Sigma'), \textbf{true}) \models (G_1 \hat{\fatsemi} G_2)^+ * \mathsf{True} \tag{5.121}$$

Besides, we get:

$$(\sigma, \Sigma') \models (Q_1 \fatsemi Q_2) \tag{5.122}$$

ii. or, $\mathbb{C} = \textbf{skip}$ and $(\theta, \Sigma) \models Q_2$.
We get:

$$(\sigma, \Sigma) \models (Q_1 \fatsemi Q_2) \tag{5.123}$$

6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \textbf{abort}$, then by the premise 1, we know: for any θ_F, $(\mathrm{C_M}, \theta \uplus \theta_F) \longrightarrow^+ \textbf{abort}$. By the premise 2 and Lemma 24, we know: $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \textbf{abort}$.

Thus we are done. □

Lemma 21. If $I \triangleright G$, $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, $(C, \sigma \uplus \sigma_F) \longrightarrow^{n+1} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

(1) either, there exist M', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \textbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$;

(2) or, there exists M' such that $M' < M$, $((\sigma, \Sigma), (\sigma', \Sigma), \textbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}, \Sigma)$.

Proof: By induction over n.
Base Case: $n = 0$. By Definition 2.
Inductive Step: $n = k + 1$. Thus there exist C_1 and σ_1' such that

$$(C, \sigma \uplus \sigma_F) \longrightarrow^1 (C_1, \sigma_1') \quad \text{and} \quad (C_1, \sigma_1') \longrightarrow^n (C', \sigma'')$$

By Definition 2, we know there exists σ_1 such that $\sigma_1' = \sigma_1 \uplus \sigma_F$ and one of the following holds:

(i) either, there exist M_1, $\mathbb{C}_1$ and Σ_1 such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}_1, \Sigma_1 \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma_1, \Sigma_1), \textbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C_1, \sigma_1, M_1) \preceq_Q (\mathbb{C}_1, \Sigma_1)$.

By the induction hypothesis, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

(a) either, there exist M', $\mathbb{C}'$ and Σ' such that $(\mathbb{C}_1, \Sigma_1 \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma_1, \Sigma_1), (\sigma', \Sigma'), \textbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$.
Then

$$(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F).$$

Since $I \triangleright G$, we know

$$((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}.$$

(b) or, there exists M' such that $M' < M_1$,
$((\sigma_1, \Sigma_1), (\sigma', \Sigma_1), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}_1, \Sigma_1)$.
Since $I \triangleright G$, we know

$$((\sigma, \Sigma), (\sigma', \Sigma_1), \mathbf{true}) \models G^+ * \mathsf{True}.$$

(ii) or, there exists M_1 such that $M_1 < M$,
$((\sigma, \Sigma), (\sigma_1, \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C_1, \sigma_1, M_1) \preceq_Q (\mathbb{C}, \Sigma)$.

The case is similar.

Thus we are done. □

Lemma 22. If $I \triangleright G$, $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, $(C, \sigma \uplus \sigma_F) \xrightarrow{e}{}^{n+1} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exist σ', M', $\mathbb{C}'$ and Σ' such that $\sigma'' = \sigma' \uplus \sigma_F$, $(\mathbb{C}, \Sigma \uplus \Sigma_F) \xrightarrow{e}{}^{+} (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', M') \preceq_Q (\mathbb{C}', \Sigma')$.

Proof: By induction over n. Similar to Lemma 21. □

Lemma 23. If $I \triangleright G$, $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$, $(C, \sigma \uplus \sigma_F) \longrightarrow^n (\mathbf{skip}, \sigma'')$ and $\Sigma \perp \Sigma_F$, then there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

(1) either, there exists Σ' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbf{skip}, \Sigma' \uplus \Sigma_F)$,
$((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma', \Sigma') \models Q$;

(2) or, $\mathbb{C} = \mathbf{skip}$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $(\sigma', \Sigma) \models Q$.

Proof: By induction over n. Similar to Lemma 21. □

Lemma 24. If $R, G, I \models (C, \sigma, M) \preceq_Q (\mathbb{C}, \Sigma)$ and $(C, \sigma \uplus \sigma_F) \longrightarrow^{n+1} \mathbf{abort}$ and $\Sigma \perp \Sigma_F$,
then $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Proof: By induction over n. Similar to Lemma 21. □

Lemma 25. If $I_1 \triangleright G_1$, $I_2 \triangleright G_2$ and $\mathsf{MPrecise}(I_1, I_2)$, then $(G_1 * \mathsf{True}) \hat{\S} (G_2 * \mathsf{True}) \Rightarrow (G_1 \hat{\S} G_2) * \mathsf{True}$.

Proof: For any σ, Σ, σ', Σ' and b, if $((\sigma, \Sigma), (\sigma', \Sigma'), b) \models (G_1 * \mathsf{True}) \hat{\S} (G_2 * \mathsf{True})$, we know there exist θ, θ', b_1 and b_2 such that

$$((\sigma, \theta), (\sigma', \theta'), b_1) \models (G_1 * \mathsf{True}), \quad ((\theta, \Sigma), (\theta', \Sigma'), b_2) \models (G_2 * \mathsf{True}), \quad b = b_1 \wedge b_2.$$

Then we know there exist σ_1, θ_1, σ_1', θ_1', θ_2, Σ_2, θ_2' and Σ_2' such that

$$((\sigma_1, \theta_1), (\sigma_1', \theta_1'), b_1) \models G_1, \quad ((\theta_2, \Sigma_2), (\theta_2', \Sigma_2'), b_2) \models G_2,$$
$$\sigma_1 \subseteq \sigma, \quad \theta_1 \subseteq \theta, \quad \sigma_1' \subseteq \sigma', \quad \theta_1' \subseteq \theta', \quad \theta_2 \subseteq \theta, \quad \Sigma_2 \subseteq \Sigma, \quad \theta_2' \subseteq \theta', \quad \Sigma_2' \subseteq \Sigma'$$

Since $I_1 \triangleright G_1$ and $I_2 \triangleright G_2$, we know

$$(\sigma_1, \theta_1) \models I_1, \quad (\sigma_1', \theta_1') \models I_1, \quad (\theta_2, \Sigma_2) \models I_2, \quad (\theta_2', \Sigma_2') \models I_2.$$

Since $\mathsf{MPrecise}(I_1, I_2)$, we know

$$\theta_1 = \theta_2, \quad \theta_1' = \theta_2'.$$

Thus we know

$$((\sigma_1, \Sigma_2), (\sigma_1', \Sigma_2'), b) \models G_1 \hat{\S} G_2$$

Thus

$$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models (G_1 \hat{\S} G_2) * \mathsf{True}.$$

Then we are done. □

Lemma 26. $(R_1 \mathbin{\tilde{\S}} R_2) * \mathsf{Id} \;\Rightarrow\; (R_1 * \mathsf{Id}) \mathbin{\tilde{\S}} (R_2 * \mathsf{Id})$.

Proof: For any σ, Σ, σ', Σ' and b, if $((\sigma, \Sigma), (\sigma', \Sigma'), b) \models (R_1 \mathbin{\tilde{\S}} R_2) * \mathsf{Id}$, we know there exist σ_1, Σ_1, σ_1', Σ_1', σ_2 and Σ_2 such that

$$\begin{gathered}((\sigma_1, \Sigma_1), (\sigma_1', \Sigma_1'), b) \models R_1 \mathbin{\tilde{\S}} R_2, \\ \sigma = \sigma_1 \uplus \sigma_2, \quad \Sigma = \Sigma_1 \uplus \Sigma_2, \quad \sigma' = \sigma_1' \uplus \sigma_2, \quad \Sigma' = \Sigma_1' \uplus \Sigma_2\end{gathered}$$

Then we know there exist θ, θ', b_1 and b_2 such that

$$((\sigma_1, \theta), (\sigma_1', \theta'), b_1) \models R_1, \quad ((\theta, \Sigma_1), (\theta', \Sigma_1'), b_2) \models R_2, \quad b = b_1 \vee b_2.$$

Thus we know

$$((\sigma, \theta), (\sigma', \theta'), b_1) \models R_1 * \mathsf{Id}, \quad ((\theta, \Sigma), (\theta', \Sigma'), b_2) \models R_2 * \mathsf{Id}.$$

Thus

$$((\sigma, \Sigma), (\sigma', \Sigma'), b) \models (R_1 * \mathsf{Id}) \mathbin{\tilde{\S}} (R_2 * \mathsf{Id}).$$

Then we are done. □

5.4 Soundness of Unary Rules

Lemma 27. If $R, G, I \vdash \{p\}C\{q\}$, then $I \triangleright \{R, G\}$, $p \vee q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(\{p, q\}, R * \mathsf{Id})$.

Proof: By induction over the derivation of $R, G, I \vdash \{p\}C\{q\}$. For the stability, we need Lemma 28. □

Lemma 28. If $\mathsf{Sta}(p, R * \mathsf{Id})$, then $\mathsf{Sta}(\lfloor p \rfloor_{\mathsf{w}}, R * \mathsf{Id})$.

Lemma 29. If $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$ and $\mathcal{H} \leq \mathcal{H}'$, then $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H}';w;q} (\mathbb{D}, \Sigma)$.

Proof: We know: if $ws' <_{\mathcal{H}} ws$ and $\mathcal{H} \leq \mathcal{H}'$, then $ws' <_{\mathcal{H}'} ws$. □

We define:

$$\mathsf{inchead}(ws, (k_1, k_2)) \stackrel{\text{def}}{=} \begin{cases} (w + k_1, n + k_2) & \text{if } ws = (w, n) \\ (w + k_1, n + k_2) :: ws' & \text{if } ws = (w, n) :: ws' \end{cases}$$

Lemma 30. If $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$, $w_1 \leq w$ and $ws_1 = \mathsf{inchead}(ws, (w_1, 0))$, then $R, G, I \models (C, \sigma, ws_1) \preceq_{\mathcal{H};w-w_1;q} (\mathbb{D}, \Sigma)$.

Proof: By co-induction. From the premise, we know: $(\sigma, \Sigma) \models I * \mathbf{true}$.

1. For any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, from the premise, we know there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:
   (a) there exist ws', w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{C}', \Sigma')$.
   By the co-induction hypothesis, let $ws'_1 = \mathsf{inchead}(ws', (w_1, 0))$, we know $R, G, I \models (C', \sigma', ws'_1) \preceq_{\mathcal{H};w'-w_1;q} (\mathbb{C}', \Sigma')$.
   (b) there exists ws' such that $ws' <_{\mathcal{H}} ws$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$.
   By the co-induction hypothesis, let $ws'_1 = \mathsf{inchead}(ws', (w_1, 0))$, we know $R, G, I \models (C', \sigma', ws'_1) \preceq_{\mathcal{H};w-w_1;q} (\mathbb{D}, \Sigma)$.
   Since $ws' <_{\mathcal{H}} ws$, we know $ws'_1 <_{\mathcal{H}} ws_1$.
2. For any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, the proof is similar to the previous case.
3. For any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$, from the premise, we know: there exist ws' and w' such that $R, G, I \models (C, \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{D}, \Sigma')$.
   By the co-induction hypothesis, let $ws'_1 = \mathsf{inchead}(ws', (w_1, 0))$, we know $R, G, I \models (C, \sigma', ws'_1) \preceq_{\mathcal{H};w'-w_1;q} (\mathbb{D}, \Sigma')$.
4. For any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$, from the premise, we know: $R, G, I \models (C, \sigma', ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma')$.
   By the co-induction hypothesis, we know $R, G, I \models (C, \sigma', ws_1) \preceq_{\mathcal{H};w-w_1;q} (\mathbb{D}, \Sigma')$.
5. If $C = \mathbf{skip}$, then for any Σ_F, if $\Sigma \perp \Sigma_F$, from the premise we know one of the following holds:
   (a) there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{C}', \Sigma') \models q$.
   (b) there exists w' such that $ws = (w', 0)$ and $(\sigma, w + w', \mathbb{D}, \Sigma) \models q$.
   Thus $ws_1 = (w' + w_1, 0)$ and $(\sigma, (w - w_1) + (w' + w_1), \mathbb{D}, \Sigma) \models q$.
6. For any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$ and $\Sigma \perp \Sigma_F$, from the premise we know: $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

The HIDE-w rule.

Lemma 31 (HIDE-w). If $R, G, I \models \{p\}C\{q\}$, then $R, G, I \models \{\lfloor p \rfloor_{\mathsf{w}}\}C\{\lfloor q \rfloor_{\mathsf{w}}\}$.

Proof: We want to prove: for all σ, w_1, $\mathbb{D}$ and Σ, if $(\sigma, w_1, \mathbb{D}, \Sigma) \models \lfloor p \rfloor_{\mathsf{w}}$, then

$$R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w_1;\lfloor q \rfloor_{\mathsf{w}}} (\mathbb{D}, \Sigma).$$

We know there exists w such that

$$(\sigma, w, \mathbb{D}, \Sigma) \models p$$

From the premise, we know:

$$R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w;q} (\mathbb{D}, \Sigma).$$

By Lemma 32, we are done. □

Lemma 32. If $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$, then $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w_1;\lfloor q \rfloor_{\mathsf{w}}} (\mathbb{D}, \Sigma)$.

Proof: By co-induction. From the premise, we know: $(\sigma, \Sigma) \models I * \mathbf{true}$.

1. For any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, from the premise, we know there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

   (a) there exist ws', w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{C}', \Sigma')$.
   By the co-induction hypothesis, we know $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w_1;\lfloor q \rfloor_{\mathsf{w}}} (\mathbb{C}', \Sigma')$.

   (b) there exists ws' such that $ws' <_{\mathcal{H}} ws$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$.
   By the co-induction hypothesis, we know $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w_1;\lfloor q \rfloor_{\mathsf{w}}} (\mathbb{D}, \Sigma)$.

2. For any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$ and $\Sigma \perp \Sigma_F$, the proof is similar to the previous case.

3. For any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$, from the premise, we know: there exist ws' and w' such that $R, G, I \models (C, \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{D}, \Sigma')$.
   By the co-induction hypothesis, we know $R, G, I \models (C, \sigma', ws') \preceq_{\mathcal{H};w_1;\lfloor q \rfloor_{\mathsf{w}}} (\mathbb{D}, \Sigma')$.

4. For any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$, from the premise, we know: $R, G, I \models (C, \sigma', ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma')$.
   By the co-induction hypothesis, we know $R, G, I \models (C, \sigma', ws) \preceq_{\mathcal{H};w_1;\lfloor q \rfloor_{\mathsf{w}}} (\mathbb{D}, \Sigma')$.

5. If $C = \mathbf{skip}$, then for any Σ_F, if $\Sigma \perp \Sigma_F$, from the premise we know one of the following holds:

   (a) there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{C}', \Sigma') \models q$.
   Thus $(\sigma, w', \mathbb{C}', \Sigma') \models \lfloor q \rfloor_{\mathsf{w}}$.

   (b) there exists w' such that $ws = (w', 0)$ and $(\sigma, w + w', \mathbb{D}, \Sigma) \models q$.
   Thus $(\sigma, w_1 + w', \mathbb{D}, \Sigma) \models \lfloor q \rfloor_{\mathsf{w}}$.

6. For any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$ and $\Sigma \perp \Sigma_F$, from the premise we know: $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

The WHILE rule.

Lemma 33 (WHILE). If

1. $R, G, I \models \{p'\}C\{p\}$;
2. $p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp})$;
3. $\mathsf{Sta}(p, R * \mathsf{Id})$; $I \triangleright \{R, G\}$; $p \Rightarrow (B = B) * I$;

then $R, G, I \models \{p\}\mathbf{while}\ (B)\ C\{p \wedge \neg B\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then

$$R, G, I \models (\mathbf{while}\ (B)\ C, \sigma, (0, |\mathbf{while}\ (B)\ C|)) \preceq_{\mathsf{height}(\mathbf{while}\ (B)\ C);w;p\wedge\neg B} (\mathbb{D}, \Sigma).$$

We know $|\mathbf{while}\ (B)\ C| = 1$ and can prove $\mathsf{height}(\mathbf{while}\ (B)\ C) = \mathsf{height}(C) + 1$.

By co-induction. From $(\sigma, w, \mathbb{D}, \Sigma) \models p$, since $p \Rightarrow I * (B = B)$, we know:

$$(\sigma, \Sigma) \models I * \mathbf{true} \tag{5.124}$$

1. For any σ_F and Σ_F, if $(\mathbf{while}\ (B)\ C, \sigma \uplus \sigma_F) \longrightarrow (C; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F)$ and $[\![B]\!]_{\sigma\uplus\sigma_F} = \mathbf{true}$, below we prove 1(b) of Definition 8 holds.

   Since $(\sigma, \Sigma) \models (B = B)$, we know $[\![B]\!]_\sigma = \mathbf{true}$. Then we know

   $$(\sigma, w, \mathbb{D}, \Sigma) \models p \wedge B \tag{5.125}$$

   Since $p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp})$, we know there exists w' such that $w' < w$ and

   $$(\sigma, w', \mathbb{D}, \Sigma) \models p' \tag{5.126}$$

   From the premise 1, we know $R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w';p} (\mathbb{D}, \Sigma)$.

   By Lemma 34, we know: let

   $$ws' = (0, 0) :: (w', |C| + 1) \tag{5.127}$$

   then

   $$R, G, I \models (C; \mathsf{while}\ (B)\{C\}, \sigma, ws') \preceq_{\mathsf{height}(C)+1;w;p\wedge\neg B} (\mathbb{D}, \Sigma) \tag{5.128}$$

   We know $ws' <_{\mathsf{height}(C)+1} (0, 1)$.

   Also, since $I \triangleright G$ and $(\sigma, \Sigma) \models I * \mathbf{true}$, we know $((\sigma, \Sigma), (\sigma, \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$.

2. For any σ_F and Σ_F, if $(\mathbf{while}\ (B)\ C, \sigma \uplus \sigma_F) \longrightarrow (\mathbf{skip}, \sigma \uplus \sigma_F)$ and $[\![B]\!]_{\sigma\uplus\sigma_F} = \mathbf{false}$, below we prove 1(b) of Definition 8 holds.

   since $(\sigma, \Sigma) \models (B = B)$, we know $[\![B]\!]_\sigma = \mathbf{false}$. Then we know

   $$(\sigma, w, \mathbb{D}, \Sigma) \models p \wedge \neg B \tag{5.129}$$

   By the SKIP and FRAME rules, we know:

   $$R, G, I \models (\mathbf{skip}, \sigma, (0, 0)) \preceq_{\mathsf{height}(C)+1;w;p\wedge\neg B} (\mathbb{D}, \Sigma) \tag{5.130}$$

   We know $(0, 0) <_{\mathsf{height}(C)+1} (0, 1)$ and $((\sigma, \Sigma), (\sigma, \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$.

3. For any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,

   since $\mathsf{Sta}(p, R * \mathsf{Id})$, we know $\mathsf{Sta}(p, R^+ * \mathsf{Id})$, thus there exists w' such that

   $$(\sigma', w', \mathbb{D}, \Sigma') \models p \tag{5.131}$$

   By the co-induction hypothesis, we get:

   $$R, G, I \models (\mathbf{while}\ (B)\ C, \sigma', (0, 1)) \preceq_{\mathsf{height}(C)+1;w';p\wedge\neg B} (\mathbb{D}, \Sigma') \tag{5.132}$$

4. For any σ' and Σ', if $((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{false}) \models R^+ * \mathsf{Id}$,
since $\mathsf{Sta}(p, R*\mathsf{Id})$, we know $\mathsf{Sta}(p, R^+*\mathsf{Id})$, thus

$$(\sigma', w, \mathbb{D}, \Sigma') \models p \tag{5.133}$$

By the co-induction hypothesis, we get:

$$R, G, I \models (\mathbf{while}\ (B)\ C, \sigma', (0,1)) \preceq_{\mathsf{height}(C)+1;w;p\wedge\neg B} (\mathbb{D}, \Sigma') \tag{5.134}$$

Thus we are done. □

Lemma 34. If

1. $R, G, I \models (C_1, \sigma, ws_1) \preceq_{\mathcal{H};w_0';p} (\mathbb{D}, \Sigma)$;
2. for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p'$, then $R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathcal{H};w;p} (\mathbb{D}, \Sigma)$;
3. $p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp})$;
4. $\mathsf{Sta}(p, R*\mathsf{Id})$; $I \triangleright \{R, G\}$; $p \Rightarrow (B = B) * I$;
5. $ws = (0,0) :: \mathsf{inchead}(ws_1, (w_0', 1))$;
6. $\mathsf{root}(ws_1) = (w_1, \_)$; $w_0' + w_1 \leq w_0$;

then $R, G, I \models (C_1; \mathsf{while}\ (B)\{C\}, \sigma, ws) \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{D}, \Sigma)$.

Proof: By co-induction. From the first premise, we know $(\sigma, \Sigma) \models I * \mathbf{true}$.

1. For any σ_F, Σ_F, C_1' and σ'', if $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow (C_1'; \mathsf{while}\ (B)\{C\}, \sigma'')$, i.e.,
$(C_1, \sigma \uplus \sigma_F) \longrightarrow (C_1', \sigma'')$, from the premise 1, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:
   (a) there exist ws_1', w_0'', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$,
   $((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C_1', \sigma', ws_1') \preceq_{\mathcal{H};w_0'';p} (\mathbb{C}', \Sigma')$.
   Suppose $\mathsf{root}(ws_1') = (w_1', \_)$.
   By the co-induction hypothesis, let $ws' = (0,0) :: \mathsf{inchead}(ws_1', (w_0'', 1))$, we know:
   $R, G, I \models (C_1'; \mathsf{while}\ (B)\{C\}, \sigma', ws') \preceq_{\mathcal{H}+1;w_0''+w_1';p\wedge\neg B} (\mathbb{C}', \Sigma')$.
   (b) there exists ws_1' such that $ws_1' <_{\mathcal{H}} ws_1$,
   $((\sigma,\Sigma),(\sigma',\Sigma),\mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C_1', \sigma', ws_1') \preceq_{\mathcal{H};w_0';p} (\mathbb{D}, \Sigma)$.
   Suppose $\mathsf{root}(ws_1') = (w_1', \_)$. Since $ws_1' <_{\mathcal{H}} ws_1$, we know $w_1' \leq w_1$. Thus $w_0' + w_1' \leq w_0$.
   By the co-induction hypothesis, let $ws' = (0,0) :: \mathsf{inchead}(ws_1', (w_0', 1))$, we know:
   $R, G, I \models (C_1'; \mathsf{while}\ (B)\{C\}, \sigma', ws') \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{D}, \Sigma)$.
   Since $ws_1' <_{\mathcal{H}} ws_1$, we know: $ws' <_{\mathcal{H}+1} ws$.
2. For any σ_F, Σ_F, e, C_1' and σ'', if $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \xrightarrow{e} (C_1'; \mathsf{while}\ (B)\{C\}, \sigma'')$, the proof is similar to the previous case.
3. For any σ_F and Σ_F, if $(C_1; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow (\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F)$, i.e., $C_1 = \mathbf{skip}$,
from the premise 1, we know one of the following holds:
   (a) there exists w_1 such that $ws_1 = (w_1, 0)$ and $(\sigma, w_1 + w_0', \mathbb{D}, \Sigma) \models p$.
   Thus $ws = (0,0) :: (w_1 + w_0', 1)$. We know $(0,0) :: (w_1 + w_0', 0) <_{\mathcal{H}+1} ws$.
   Also we know $((\sigma,\Sigma),(\sigma,\Sigma),\mathbf{false}) \models G^+ * \mathsf{True}$.
   Below we prove:

$$R, G, I \models (\mathsf{while}\ (B)\{C\}, \sigma, (0,0) :: (w_1 + w_0', 0)) \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{C}', \Sigma') \tag{5.135}$$

   By co-induction. Since $p \Rightarrow I * (B = B)$, we know $(\sigma, \Sigma') \models I * \mathbf{true}$.

i. For any σ_F and Σ_F, if $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow (C; \mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F)$ and $[\![B]\!]_{\sigma \uplus \sigma_F} = \mathbf{true}$, below we prove 1(b) of Definition 8 holds.
Since $(\sigma, \Sigma') \models (B = B)$, we know $[\![B]\!]_\sigma = \mathbf{true}$. Then we know

$$(\sigma, w_1 + w_0', \mathbb{C}', \Sigma') \models p \wedge B \tag{5.136}$$

Since $p \wedge B \Rightarrow p' * (\mathsf{wf}(1) \wedge \mathsf{emp})$, we know there exists w_1' such that $w_1' < w_1 + w_0'$ and

$$(\sigma, w_1', \mathbb{C}', \Sigma') \models p' \tag{5.137}$$

From the premise 2, we know $R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathcal{H};w_1';p} (\mathbb{C}', \Sigma')$.
By the co-induction hypothesis, we know:

$$R, G, I \models (C; \mathsf{while}\ (B)\{C\}, \sigma, (0,0) :: (w_1', |C|+1)) \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{C}', \Sigma') \tag{5.138}$$

We know $(0,0) :: (w_1', |C|+1) <_{\mathcal{H}+1} (0,0) :: (w_1 + w_0', 0)$.
Also we know $((\sigma, \Sigma'), (\sigma, \Sigma'), \mathbf{false}) \models G^+ * \mathsf{True}$.

ii. For any σ_F and Σ_F, if $(\mathsf{while}\ (B)\{C\}, \sigma \uplus \sigma_F) \longrightarrow (\mathbf{skip}, \sigma \uplus \sigma_F)$ and $[\![B]\!]_{\sigma \uplus \sigma_F} = \mathbf{false}$, below we prove 1(b) of Definition 8 holds.
Since $(\sigma, \Sigma') \models (B = B)$, we know $[\![B]\!]_\sigma = \mathbf{false}$. Since $(\sigma, w_1 + w_0', \mathbb{C}', \Sigma') \models p$, we know:

$$(\sigma, w_1 + w_0', \mathbb{C}', \Sigma') \models p \wedge \neg B \tag{5.139}$$

Since $w_1 + w_0' \leq w_0$, we know:

$$(\sigma, w_0, \mathbb{C}', \Sigma') \models p \wedge \neg B \tag{5.140}$$

By the SKIP and FRAME rules, we know:

$$R, G, I \models (\mathbf{skip}, \sigma, (0,0)) \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{C}', \Sigma') \tag{5.141}$$

We know $(0,0) <_{\mathcal{H}+1} (0,0) :: (w_1 + w_0', 0)$ and $((\sigma, \Sigma'), (\sigma, \Sigma'), \mathbf{false}) \models G^+ * \mathsf{True}$.

iii. For any σ' and Σ'', if $((\sigma, \Sigma'), (\sigma', \Sigma''), \mathbf{true}) \models R^+ * \mathsf{Id}$,
since $\mathsf{Sta}(p, R * \mathsf{Id})$, we know $\mathsf{Sta}(p, R^+ * \mathsf{Id})$, thus there exists w_1' such that

$$(\sigma', w_1' + w_0', \mathbb{C}', \Sigma'') \models p \tag{5.142}$$

By the co-induction hypothesis, we get:

$$R, G, I \models (\mathsf{while}\ (B)\{C\}, \sigma', (0,0) :: (w_1' + w_0', 0)) \preceq_{\mathcal{H}+1;w_1'+w_0';p\wedge\neg B} (\mathbb{C}', \Sigma'') \tag{5.143}$$

iv. For any σ' and Σ'', if $((\sigma, \Sigma'), (\sigma', \Sigma''), \mathbf{false}) \models R^+ * \mathsf{Id}$,
since $\mathsf{Sta}(p, R * \mathsf{Id})$, we know $\mathsf{Sta}(p, R^+ * \mathsf{Id})$, thus

$$(\sigma', w_1 + w_0', \mathbb{C}', \Sigma'') \models p \tag{5.144}$$

By the co-induction hypothesis, we get:

$$R, G, I \models (\mathsf{while}\ (B)\{C\}, \sigma', (0,0) :: (w_1 + w_0', 0)) \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{C}', \Sigma'') \tag{5.145}$$

Thus we have proved (5.135).

(b) there exist w_1', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w_1', \mathbb{C}', \Sigma') \models p$.
We can prove:

$$R, G, I \models (\mathsf{while}\ (B)\{C\}, \sigma, (0,0) :: (w_1', 0)) \preceq_{\mathcal{H}+1;w_1';p\wedge\neg B} (\mathbb{C}', \Sigma') \tag{5.146}$$

in the similar way as the previous case.

4. For any σ' and Σ', if $((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{true}) \models R^+ * \mathsf{Id}$,
   from the premise, we know there exist ws_1' and w_0'' such that $R,G,I \models (C_1,\sigma',ws_1') \preceq_{\mathcal{H};w_0'';p} (\mathbb{D},\Sigma')$.
   Suppose $\mathsf{root}(ws_1') = (w_1',\_)$.
   By the co-induction hypothesis, we know: let $ws' = (0,0) :: \mathsf{inchead}(ws_1',(w_0'',1))$, then $R,G,I \models (C_1;\mathsf{while}\ (B)\{C\},\sigma',ws') \preceq_{\mathcal{H}+1;w_0''+w_1';p\wedge\neg B} (\mathbb{D},\Sigma')$.
5. For any σ' and Σ', if $((\sigma,\Sigma),(\sigma',\Sigma'),\mathbf{false}) \models R^+ * \mathsf{Id}$,
   from the premise, we know: $R,G,I \models (C_1,\sigma',ws_1) \preceq_{\mathcal{H};w_0';p} (\mathbb{D},\Sigma')$.
   By the co-induction hypothesis, we know: $R,G,I \models (C_1;\mathsf{while}\ (B)\{C\},\sigma',ws) \preceq_{\mathcal{H}+1;w_0;p\wedge\neg B} (\mathbb{D},\Sigma')$.
6. For any σ_F and Σ_F, if $(C_1;\mathsf{while}\ (B)\{C\},\sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$, we know $(C_1,\sigma\uplus\sigma_F) \longrightarrow \mathbf{abort}$. By the premise 1, we know: $(\mathbb{D},\Sigma\uplus\Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

The SEQ rule.

Lemma 35 (SEQ). If

1. $R,G,I \models \{p\}C_1\{p'\}$;
2. $R,G,I \models \{p'\}C_2\{q\}$;
3. $I \triangleright G$;

then $R,G,I \models \{p\}C_1;C_2\{q\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma,w,\mathbb{D},\Sigma) \models p$, then

$$R,G,I \models (C_1;C_2,\sigma,(0,|C_1;C_2|)) \preceq_{\mathsf{height}(C_1;C_2);w;q} (\mathbb{D},\Sigma).$$

We know $|C_1;C_2| = |C_1| + |C_2| + 1$ and can prove $\mathsf{height}(C_1;C_2) = max\{\mathsf{height}(C_1),\mathsf{height}(C_2)\}$.
Since $(\sigma,w,\mathbb{D},\Sigma) \models p$, by the premise 1, we know:

$$R,G,I \models (C_1,\sigma,(0,|C_1|)) \preceq_{\mathsf{height}(C_1);w;p'} (\mathbb{D},\Sigma).$$

By Lemma 29, we know: $R,G,I \models (C_1,\sigma,(0,|C_1|)) \preceq_{\mathsf{height}(C_1;C_2);w;p'} (\mathbb{D},\Sigma)$.
From the premise 2, by Lemma 29, we know: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma,w,\mathbb{D},\Sigma) \models p'$, then $R,G,I \models (C_2,\sigma,(0,|C_2|)) \preceq_{\mathsf{height}(C_1;C_2);w;q} (\mathbb{D},\Sigma)$.
By Lemma 36, we are done. □

Lemma 36. If

1. $R,G,I \models (C_1,\sigma,ws_1) \preceq_{\mathcal{H};w;p'} (\mathbb{D},\Sigma)$;
2. for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma,w,\mathbb{D},\Sigma) \models p'$, then $R,G,I \models (C_2,\sigma,(0,|C_2|)) \preceq_{\mathcal{H};w;q} (\mathbb{D},\Sigma)$;
3. $I \triangleright G$;
4. $ws = \mathsf{inchead}(ws_1,(0,|C_2|+1))$;

then $R,G,I \models (C_1;C_2,\sigma,ws) \preceq_{\mathcal{H};w;q} (\mathbb{D},\Sigma)$.

Proof: By co-induction. From the premise 1, we know: $(\sigma,\Sigma) \models I * \mathbf{true}$.

1. for any σ_F, Σ_F, C_1' and σ'', if $(C_1;C_2,\sigma\uplus\sigma_F) \longrightarrow (C_1';C_2,\sigma'')$, i.e., $(C_1,\sigma\uplus\sigma_F) \longrightarrow (C_1',\sigma'')$,
   from the premise 1, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

(a) there exist ws'_1, w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C'_1, \sigma', ws'_1) \preceq_{\mathcal{H};w';p'} (\mathbb{C}', \Sigma')$.
By the co-induction hypothesis, we know: let $ws' = \mathsf{inchead}(ws'_1, (0, |C_2| + 1))$, then $R, G, I \models (C'_1; C_2, \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{C}', \Sigma')$.

(b) there exists ws'_1 such that $ws'_1 <_{\mathcal{H}} ws_1$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C'_1, \sigma', ws'_1) \preceq_{\mathcal{H};w;p'} (\mathbb{D}, \Sigma)$.
By the co-induction hypothesis, we know: let $ws' = \mathsf{inchead}(ws'_1, (0, |C_2| + 1))$, $R, G, I \models (C'_1; C_2, \sigma', ws') \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$.
Since $ws'_1 <_{\mathcal{H}} ws_1$, we know: $ws' <_{\mathcal{H}} ws$.

2. for any σ_F, Σ_F, e, C'_1 and σ'', if $(C_1; C_2, \sigma \uplus \sigma_F) \xrightarrow{e} (C'_1; C_2, \sigma'')$, the proof is similar to the previous case.

3. for any σ_F and Σ_F, if $(C_1; C_2, \sigma \uplus \sigma_F) \longrightarrow (C_2, \sigma \uplus \sigma_F)$ and $C_1 = \mathbf{skip}$,
from the premise 1, we know one of the following holds:

(a) there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{C}', \Sigma') \models p'$.
From the premise 2, we know: $R, G, I \models (C_2, \sigma, (0, |C_2|)) \preceq_{\mathcal{H};w';q} (\mathbb{C}', \Sigma')$.

(b) there exists w_1 such that $ws_1 = (w_1, 0)$ and $(\sigma, w + w_1, \mathbb{D}, \Sigma) \models p'$.
Thus we know $ws = (w_1, |C_2| + 1)$.
We know $(w_1, |C_2|) <_{\mathcal{H}} ws$.
Since $(\sigma, \Sigma) \models I * \mathbf{true}$ and $I \triangleright G$, we know $((\sigma, \Sigma), (\sigma, \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$.
From the premise 2, we know: $R, G, I \models (C_2, \sigma, (0, |C_2|)) \preceq_{\mathcal{H};w+w_1;q} (\mathbb{D}, \Sigma)$.
By Lemma 30, we get: $R, G, I \models (C_2, \sigma, (w_1, |C_2|)) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$.

4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,
from the premise, we know: there exists ws'_1 and w' such that $R, G, I \models (C_1, \sigma', ws'_1) \preceq_{\mathcal{H};w';p'} (\mathbb{D}, \Sigma')$.
By the co-induction hypothesis, we know: let $ws' = \mathsf{inchead}(ws'_1, (0, |C_2| + 1))$, then $R, G, I \models (C_1; C_2, \sigma', ws') \preceq_{\mathcal{H};w';q} (\mathbb{D}, \Sigma')$.

5. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$,
from the premise, we know: $R, G, I \models (C_1, \sigma', ws_1) \preceq_{\mathcal{H};w;p'} (\mathbb{D}, \Sigma')$.
By the co-induction hypothesis, we know: $R, G, I \models (C_1; C_2, \sigma', ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma')$.

6. for any σ_F and Σ_F, if $(C_1; C_2, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$, we know: $(C_1, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$. By the premise 1, we know: $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

The ATOM rule.

Lemma 37 (ATOM). If

1. $\models_{\text{SL}} [p]C[q]$;
2. $(\llbracket p \rrbracket \ltimes \llbracket q \rrbracket) \Rightarrow G * \mathsf{True}$;
3. $p \vee q \Rightarrow I * \mathbf{true}$;
4. $\mathsf{Locality}(C)$;

then $[I], G, I \models \{p\}\langle C\rangle\{q\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then

$$[I], G, I \models (\langle C\rangle, \sigma, (0, |\langle C\rangle|)) \preceq_{\mathsf{height}(\langle C\rangle);w;q} (\mathbb{D}, \Sigma).$$

We know $|\langle C\rangle| = 1$ and can prove $\mathsf{height}(\langle C\rangle) = 1$.

By co-induction. Since $p \Rightarrow I * \mathbf{true}$, we know $(\sigma, \Sigma) \models I * \mathbf{true}$. From the premises 1 and 2, we can prove:

$$(C, \sigma) \not\longrightarrow^* \mathbf{abort}\,, \quad (C, \sigma) \not\longrightarrow^\omega . \tag{5.147}$$

By $\mathsf{Locality}(C)$, we know: for any σ_F,

$$(C, \sigma \uplus \sigma_F) \not\longrightarrow^* \mathbf{abort}\,, \quad (C, \sigma \uplus \sigma_F) \not\longrightarrow^\omega . \tag{5.148}$$

1. for any σ_F, Σ_F, C' and σ'', if $(\langle C\rangle, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,
   by the operational semantics, we know $C' = \mathbf{skip}$ and

   $$(C, \sigma \uplus \sigma_F) \longrightarrow^* (\mathbf{skip}, \sigma'') \tag{5.149}$$

   by $\mathsf{Locality}(C)$, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$.
   From $\models_{\text{SL}} [p]C[q]$ and $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$, we know:

   $$(\sigma', w, \mathbb{D}, \Sigma) \models q \tag{5.150}$$

   Thus we know:

   $$((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models [\![p]\!] \ltimes [\![q]\!] \tag{5.151}$$

   Since $([\![p]\!] \ltimes [\![q]\!]) \Rightarrow G * \mathsf{True}$, we know $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$.
   Since $q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(q, [I] * \mathsf{Id})$, by the SKIP and FRAME rules, we know:

   $$[I], G, I \models (\mathbf{skip}, \sigma', (0, 0)) \preceq_{1;w;q} (\mathbb{D}, \Sigma) \tag{5.152}$$

   Also, we know: $(0, 0) <_1 (0, 1)$.
2. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models ([I])^+ * \mathsf{Id}$, we know $\sigma' = \sigma$ and $\Sigma' = \Sigma$.
   By the co-induction hypothesis, we know: $[I], G, I \models (\langle C\rangle, \sigma, (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma)$.
3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models ([I])^+ * \mathsf{Id}$, we know $\sigma' = \sigma$ and $\Sigma' = \Sigma$.
   By the co-induction hypothesis, we know: $[I], G, I \models (\langle C\rangle, \sigma, (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma)$.

Thus we are done. □

The ATOM$^+$ rule.

Lemma 38 (ATOM$^+$). If

1. $\models_{\text{SL}} [p']C[q']$;
2. $p \Rrightarrow^a p'$; $q' \Rrightarrow^b q$; $+ \in \{a, b\}$;
3. $([\![p]\!] \propto [\![q]\!]) \Rightarrow G * \mathsf{True}$;
4. $p \vee q \Rightarrow I * \mathbf{true}$;
5. $\mathsf{Locality}(C)$;

then $[I], G, I \models \{p\}\langle C\rangle\{q\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then

$$[I], G, I \models (\langle C\rangle, \sigma, (0, |\langle C\rangle|)) \preceq_{\mathsf{height}(\langle C\rangle);w;q} (\mathbb{D}, \Sigma).$$

We know $|\langle C\rangle| = 1$ and can prove $\mathsf{height}(\langle C\rangle) = 1$.

By co-induction. Since $p \Rightarrow I * \mathbf{true}$, we know $(\sigma, \Sigma) \models I * \mathbf{true}$. From the premises 1 and 2, we can prove:

$$(C, \sigma) \not\longrightarrow^* \mathbf{abort}\,, \quad (C, \sigma) \not\longrightarrow^\omega . \tag{5.153}$$

By $\mathsf{Locality}(C)$, we know: for any σ_F,

$$(C, \sigma \uplus \sigma_F) \not\longrightarrow^* \mathbf{abort}\,, \quad (C, \sigma \uplus \sigma_F) \not\longrightarrow^\omega . \tag{5.154}$$

1. for any σ_F, Σ_F, C' and σ'', if $(\langle C\rangle, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,
   by the operational semantics, we know $C' = \mathbf{skip}$ and

   $$(C, \sigma \uplus \sigma_F) \longrightarrow^* (\mathbf{skip}, \sigma'') \tag{5.155}$$

   by $\mathsf{Locality}(C)$, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$.
   From $p \Rrightarrow^a p'$, we know one of the following holds:

   (a) either, a is $+$, and there exist w', $\mathbb{D}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}', \Sigma' \uplus \Sigma_F)$ and $(\sigma, w', \mathbb{D}', \Sigma') \models p'$;

   (b) or, a is 0, and there exist w', $\mathbb{D}'$ and Σ' such that $(\sigma, w', \mathbb{D}', \Sigma') \models p'$, $w' = w$, $\mathbb{D}' = \mathbb{D}$ and $\Sigma' = \Sigma$.

   For either case, from $\models_{\text{SL}} [p']C[q']$ and $(C, \sigma) \longrightarrow^* (\mathbf{skip}, \sigma')$, we know:

   $$(\sigma', w', \mathbb{D}', \Sigma') \models q' \tag{5.156}$$

   From $q' \Rrightarrow^b q$, we know one of the following holds:

   (a) either, b is $+$, and there exist w'', $\mathbb{D}''$ and Σ'' such that $(\mathbb{D}', \Sigma' \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}'', \Sigma'' \uplus \Sigma_F)$ and $(\sigma', w'', \mathbb{D}'', \Sigma'') \models q$;

   (b) or, b is 0, and there exist w'', $\mathbb{D}''$ and Σ'' such that $(\sigma', w'', \mathbb{D}'', \Sigma'') \models q$, $w'' = w'$, $\mathbb{D}'' = \mathbb{D}'$ and $\Sigma'' = \Sigma'$.

   Since $+ \in \{a, b\}$, we know the following must hold:
   there exist w'', $\mathbb{C}''$ and Σ'' such that $(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}'', \Sigma'' \uplus \Sigma_F)$ and $(\sigma', w'', \mathbb{C}'', \Sigma'') \models q$.
   We know:

   $$((\sigma, \Sigma), (\sigma', \Sigma''), \mathbf{true}) \models [\![p]\!] \propto [\![q]\!] \tag{5.157}$$

   Since $([\![p]\!] \propto [\![q]\!]) \Rightarrow G * \mathsf{True}$, we know $((\sigma, \Sigma), (\sigma', \Sigma''), \mathbf{true}) \models G^+ * \mathsf{True}$.
   Since $q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(q, [I] * \mathsf{Id})$, by the SKIP and FRAME rules, we know:

   $$[I], G, I \models (\mathbf{skip}, \sigma', (0, 0)) \preceq_{1;w';q} (\mathbb{C}'', \Sigma'') \tag{5.158}$$

2. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models ([I])^+ * \mathsf{Id}$, we know $\sigma' = \sigma$ and $\Sigma' = \Sigma$.
   By the co-induction hypothesis, we know: $[I], G, I \models (\langle C\rangle, \sigma, (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma)$.

3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models ([I])^+ * \mathsf{Id}$, we know $\sigma' = \sigma$ and $\Sigma' = \Sigma$.
   By the co-induction hypothesis, we know: $[I], G, I \models (\langle C\rangle, \sigma, (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma)$.

Thus we are done. □

Lemma 39. If

1. $R, G, I \vdash \{p\}\langle C\rangle\{q\}$;
2. $\vdash_{\text{SL}}$ is sound w.r.t. $\models_{\text{SL}}$;
3. $\mathsf{Locality}(C)$;
4. $(\sigma, w, \mathbb{D}, \Sigma) \models p$,

then for any σ_F, $(C, \sigma \uplus \sigma_F) \not\rightarrow^* \textbf{abort}$ and $(C, \sigma \uplus \sigma_F) \not\rightarrow^\omega \cdot$.

Proof: By induction over the derivation of $R, G, I \vdash \{p\}\langle C\rangle\{q\}$. □

The ATOM-R rule.

Lemma 40 (ATOM-R). If

1. $[I], G, I \models \{p\}\langle C\rangle\{q\}$;
2. $\mathsf{Sta}(\{p, q\}, R * \mathsf{Id})$; $I \triangleright \{R, G\}$; $p \vee q \Rightarrow I * \textbf{true}$;
3. for all σ and σ_F, if $(\sigma, \_, \_, \_, \_) \models p$, $(C, \sigma \uplus \sigma_F) \not\rightarrow^* \textbf{abort}$ and $(C, \sigma \uplus \sigma_F) \not\rightarrow^\omega \cdot$;

then $R, G, I \models \{p\}\langle C\rangle\{q\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then

$$R, G, I \models (\langle C\rangle, \sigma, (0, |\langle C\rangle|)) \preceq_{\mathsf{height}(\langle C\rangle);w;q} (\mathbb{D}, \Sigma).$$

We know $|\langle C\rangle| = 1$ and can prove $\mathsf{height}(\langle C\rangle) = 1$.

By co-induction. Since $p \Rightarrow I * \textbf{true}$, we know $(\sigma, \Sigma) \models I * \textbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(\langle C\rangle, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,
   by the operational semantics, we know $C' = \textbf{skip}$ and
   $$(C, \sigma \uplus \sigma_F) \longrightarrow^* (\textbf{skip}, \sigma'') \tag{5.159}$$
   From the first premise, we know:
   $$[I], G, I \models (\langle C\rangle, \sigma, (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma).$$
   Thus there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:
   (a) there exist ws', w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma', \Sigma'), \textbf{true}) \models G^+ * \mathsf{True}$ and
   $$[I], G, I \models (\textbf{skip}, \sigma', ws') \preceq_{1;w';q} (\mathbb{C}', \Sigma') \tag{5.160}$$
   From (5.160), we know one of the following holds:
   i. there exist w'', $\mathbb{C}''$ and Σ'' such that $(\mathbb{C}', \Sigma' \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}'', \Sigma'' \uplus \Sigma_F)$, $((\sigma', \Sigma'), (\sigma', \Sigma''), \textbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma', w'', \mathbb{C}'', \Sigma'') \models q$.
   Thus we know:
   $$(\mathbb{C}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}'', \Sigma'' \uplus \Sigma_F) \tag{5.161}$$
   $$((\sigma, \Sigma), (\sigma', \Sigma''), \textbf{true}) \models G^+ * \mathsf{True} \tag{5.162}$$
   Since $q \Rightarrow I * \textbf{true}$ and $\mathsf{Sta}(q, R * \mathsf{Id})$, by the SKIP and FRAME rules, we know:
   $$R, G, I \models (\textbf{skip}, \sigma', (0, 0)) \preceq_{1;w'';q} (\mathbb{C}'', \Sigma'') \tag{5.163}$$

ii. there exists w'' such that $ws' = (w'', 0)$ and $(\sigma', w' + w'', \mathbb{C}', \Sigma') \models q$.
Since $q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(q, R * \mathsf{Id})$, by the SKIP and FRAME rules, we know:

$$R, G, I \models (\mathbf{skip}, \sigma', (0,0)) \preceq_{1;w'+w'';q} (\mathbb{C}', \Sigma') \tag{5.164}$$

(b) there exists ws' such that $ws' <_1 (0,1)$, $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and

$$[I], G, I \models (\mathbf{skip}, \sigma', ws') \preceq_{1;w;q} (\mathbb{D}, \Sigma) \tag{5.165}$$

From (5.165), we know one of the following holds:

i. there exist w', $\mathbb{C}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}', \Sigma' \uplus \Sigma_F)$, $((\sigma', \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma', w', \mathbb{C}', \Sigma') \models q$.
Thus we know:

$$((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True} \tag{5.166}$$

Since $q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(q, R * \mathsf{Id})$, by the SKIP and FRAME rules, we know:

$$R, G, I \models (\mathbf{skip}, \sigma', (0,0)) \preceq_{1;w';q} (\mathbb{C}', \Sigma') \tag{5.167}$$

ii. there exists w' such that $ws' = (w', 0)$ and $(\sigma', w + w', \mathbb{D}, \Sigma) \models q$.
Since $ws' <_1 (0,1)$, we know $w' = 0$.
Since $q \Rightarrow I * \mathbf{true}$ and $\mathsf{Sta}(q, R * \mathsf{Id})$, by the SKIP and FRAME rules, we know:

$$R, G, I \models (\mathbf{skip}, \sigma', (0,0)) \preceq_{1;w;q} (\mathbb{D}, \Sigma) \tag{5.168}$$

2. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,
Since $(\sigma, w, \mathbb{D}, \Sigma) \models p$ and $\mathsf{Sta}(p, R * \mathsf{Id})$, we know there exists w' such that $(\sigma', w', \mathbb{D}, \Sigma') \models p$.
By the co-induction hypothesis, we know: $R, G, I \models (\langle C \rangle, \sigma', (0,1)) \preceq_{1;w';q} (\mathbb{D}, \Sigma')$.

3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$,
Since $(\sigma, w, \mathbb{D}, \Sigma) \models p$ and $\mathsf{Sta}(p, R * \mathsf{Id})$, we know $(\sigma', w, \mathbb{D}, \Sigma') \models p$.
By the co-induction hypothesis, we know: $R, G, I \models (\langle C \rangle, \sigma', (0,1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma')$.

Thus we are done. □

The A-CONSEQ rule.

Lemma 41 (A-CONSEQ). If

1. $p \overset{G}{\Rrightarrow} p'$;
2. $R, G, I \models \{p'\} C \{q'\}$;
3. $q' \overset{G}{\Rrightarrow} q$;
4. $\mathsf{Sta}(\{p, q\}, R * \mathsf{Id})$; $I \triangleright \{R, G\}$; $p \vee q \vee p' \vee q' \Rightarrow I * \mathbf{true}$;

then $R, G, I \models \{p\} C \{q\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then

$$R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w;q} (\mathbb{D}, \Sigma).$$

Let $\mathcal{H} = \mathsf{height}(C)$.
By co-induction. Since $p \Rightarrow I * \mathbf{true}$, we know $(\sigma, \Sigma) \models I * \mathbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,

   from $p \overset{G}{\Rightarrow} p'$, we know one of the following holds:

   (a) either, there exist w', $\mathbb{D}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}', \Sigma' \uplus \Sigma_F)$
   $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{D}', \Sigma') \models p'$;

   (b) or, there exist w', $\mathbb{D}'$ and Σ' such that $(\sigma, w', \mathbb{D}', \Sigma') \models p'$, $w' = w$, $\mathbb{D}' = \mathbb{D}$ and $\Sigma' = \Sigma$.

   For either case, from $R, G, I \models \{p'\}C\{q'\}$, we know:

   $$R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathcal{H};w';q'} (\mathbb{D}', \Sigma') \tag{5.169}$$

   Thus there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and one of the following holds:

   (a) either, there exist ws', w'', $\mathbb{C}''$ and Σ'' such that $(\mathbb{D}', \Sigma' \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}'', \Sigma'' \uplus \Sigma_F)$,
   $((\sigma, \Sigma'), (\sigma', \Sigma''), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w'';q'} (\mathbb{C}'', \Sigma'')$;

   (b) or, there exists ws' such that $ws' <_{\mathcal{H}} (0, |C|)$,
   $((\sigma, \Sigma'), (\sigma', \Sigma'), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w';q'} (\mathbb{D}', \Sigma')$.

   Then, we know one of the following holds:

   (a) there exist ws', w'', $\mathbb{C}''$ and Σ'' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}'', \Sigma'' \uplus \Sigma_F)$,
   $((\sigma, \Sigma), (\sigma', \Sigma''), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w'';q'} (\mathbb{C}'', \Sigma'')$.
   By Lemma 42, we know:

   $$R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w'';q} (\mathbb{C}'', \Sigma'') \tag{5.170}$$

   (b) there exists ws' such that $ws' <_{\mathcal{H}} (0, |C|)$,
   $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w;q'} (\mathbb{D}, \Sigma)$.
   By Lemma 42, we know:

   $$R, G, I \models (C', \sigma', ws') \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma) \tag{5.171}$$

2. for any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$, the proof is similar to the previous case.

3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,
   Since $(\sigma, w, \mathbb{D}, \Sigma) \models p$ and $\mathsf{Sta}(p, R * \mathsf{Id})$, we know there exists w' such that $(\sigma', w', \mathbb{D}, \Sigma') \models p$.
   By the co-induction hypothesis, we know: $R, G, I \models (C, \sigma', (0, |C|)) \preceq_{\mathcal{H};w';q} (\mathbb{D}, \Sigma')$.

4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$,
   Since $(\sigma, w, \mathbb{D}, \Sigma) \models p$ and $\mathsf{Sta}(p, R * \mathsf{Id})$, we know $(\sigma', w, \mathbb{D}, \Sigma') \models p$.
   By the co-induction hypothesis, we know: $R, G, I \models (C, \sigma', (0, |C|)) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma')$.

5. if $C = \mathbf{skip}$, then for any Σ_F,

   from $p \overset{G}{\Rightarrow} p'$, we know one of the following holds:

   (a) either, there exist w', $\mathbb{D}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}', \Sigma' \uplus \Sigma_F)$
   $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{D}', \Sigma') \models p'$;

   (b) or, there exist w', $\mathbb{D}'$ and Σ' such that $(\sigma, w', \mathbb{D}', \Sigma') \models p'$, $w' = w$, $\mathbb{D}' = \mathbb{D}$ and $\Sigma' = \Sigma$.

   For either case, from $R, G, I \models \{p'\}C\{q'\}$, we know:

   $$R, G, I \models (\mathbf{skip}, \sigma, (0, 0)) \preceq_{\mathcal{H};w';q'} (\mathbb{D}', \Sigma') \tag{5.172}$$

   Then one of the following holds:

(a) either, there exist w'', $\mathbb{D}''$ and Σ'' such that $(\mathbb{D}', \Sigma' \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}'', \Sigma'' \uplus \Sigma_F)$, $((\sigma, \Sigma'), (\sigma, \Sigma''), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w'', \mathbb{D}'', \Sigma'') \models q'$;

(b) or, there exist w'', $\mathbb{D}''$ and Σ'' such that $w'' = w'$, $\mathbb{D}'' = \mathbb{D}'$, $\Sigma'' = \Sigma'$ and $(\sigma, w'', \mathbb{D}'', \Sigma'') \models q'$.

From $q' \overset{G}{\Rrightarrow} q$, we know one of the following holds:

(a) either, there exist w''', $\mathbb{D}'''$ and Σ''' such that $(\mathbb{D}'', \Sigma'' \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}''', \Sigma''' \uplus \Sigma_F)$ $((\sigma, \Sigma''), (\sigma, \Sigma'''), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w''', \mathbb{D}''', \Sigma''') \models q$;

(b) or, there exist w''', $\mathbb{D}'''$ and Σ''' such that $(\sigma, w''', \mathbb{D}''', \Sigma''') \models q$, $w''' = w''$, $\mathbb{D}''' = \mathbb{D}''$ and $\Sigma''' = \Sigma''$.

Thus we get one of the following holds:

(a) either, there exist w''', $\mathbb{C}'''$ and Σ''' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}''', \Sigma''' \uplus \Sigma_F)$ $((\sigma, \Sigma), (\sigma, \Sigma'''), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w''', \mathbb{C}''', \Sigma''') \models q$;

(b) or, $(\sigma, w, \mathbb{D}, \Sigma) \models q$.

6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$,

   from $p \overset{G}{\Rrightarrow} p'$, we know one of the following holds:

   (a) either, there exist w', $\mathbb{D}'$ and Σ' such that $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{D}', \Sigma' \uplus \Sigma_F)$ $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w', \mathbb{D}', \Sigma') \models p'$;

   (b) or, there exist w', $\mathbb{D}'$ and Σ' such that $(\sigma, w', \mathbb{D}', \Sigma') \models p'$, $w' = w$, $\mathbb{D}' = \mathbb{D}$ and $\Sigma' = \Sigma$.

   For either case, from $R, G, I \models \{p'\}C\{q'\}$, we know:

$$R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathcal{H};w';q'} (\mathbb{D}', \Sigma') \tag{5.173}$$

   Then we know: $(\mathbb{D}', \Sigma' \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$. Thus $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

Lemma 42. If

1. $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;q'} (\mathbb{D}, \Sigma)$;
2. $q' \overset{G}{\Rrightarrow} q$;
3. $\mathsf{Sta}(q, R * \mathsf{Id})$; $I \triangleright \{R, G\}$; $q \Rightarrow I * \mathbf{true}$;

then $R, G, I \models (C, \sigma, ws) \preceq_{\mathcal{H};w;q} (\mathbb{D}, \Sigma)$.

Proof: By co-induction. □

The ENV rule.

Lemma 43 (ENV). If $\models_{\text{SL}} [p]c[q]$, c is silent and $\mathsf{Locality}(c)$, then $\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models \{p\}c\{q\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p$, then

$$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models (c, \sigma, (0, |c|)) \preceq_{\mathsf{height}(c);w;q} (\mathbb{D}, \Sigma).$$

We know $|c| = 1$ and can prove $\mathsf{height}(c) = 1$.

By co-induction. We know $(\sigma, \Sigma) \models \mathsf{emp} * \mathbf{true}$. From $\models_{\text{SL}} [p]c[q]$, we know:

$$(c, \sigma) \not\longrightarrow^* \mathbf{abort}\,, \quad (c, \sigma) \not\longrightarrow^\omega . \tag{5.174}$$

By $\mathsf{Locality}(c)$, we know: for any σ_F,

$$(c, \sigma \uplus \sigma_F) \not\longrightarrow^* \mathbf{abort}\,, \quad (c, \sigma \uplus \sigma_F) \not\longrightarrow^\omega . \tag{5.175}$$

1. for any σ_F, Σ_F, C' and σ'', if $(c, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,
   by the operational semantics, we know $C' = \mathbf{skip}$.
   By $\mathsf{Locality}(c)$, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$ and $(c, \sigma) \longrightarrow (\mathbf{skip}, \sigma')$.
   From $\models_{\text{SL}} [p]c[q]$, we know:
   $$(\sigma', w, \mathbb{D}, \Sigma) \models q \tag{5.176}$$
   By the SKIP rule, we know:
   $$\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models (\mathbf{skip}, \sigma', (0, 0)) \preceq_{1;w;q} (\mathbb{D}, \Sigma) \tag{5.177}$$
   We know $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models \mathsf{Emp}^+ * \mathsf{True}$.
   Also, we know: $(0, 0) <_1 (0, 1)$.
2. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models \mathsf{Emp}^+ * \mathsf{Id}$, we know $\sigma' = \sigma$ and $\Sigma' = \Sigma$.
   By the co-induction hypothesis, we know: $\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models (c, \sigma', (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma')$.
3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models \mathsf{Emp}^+ * \mathsf{Id}$, we know $\sigma' = \sigma$ and $\Sigma' = \Sigma$.
   By the co-induction hypothesis, we know: $\mathsf{Emp}, \mathsf{Emp}, \mathsf{emp} \models (c, \sigma', (0, 1)) \preceq_{1;w;q} (\mathbb{D}, \Sigma')$.

Thus we are done. □

The FRAME rule.

Lemma 44 (FRAME). If

1. $R, G, I \models \{p\}C\{q\}$;
2. $\mathsf{Sta}(\{p, q\}, R * \mathsf{Id})$; $\mathsf{Sta}(p', (R')^+ * \mathsf{Id})$; $I \triangleright \{R, G\}$; $I' \triangleright \{R', G'\}$; $p \vee q \Rightarrow I * \mathbf{true}$; $p' \Rightarrow I' * \mathbf{true}$; $G^+ \Rightarrow G$;

then $R * R', G * G', I * I' \models \{p * p'\}C\{q * p'\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p * p'$, then

$$R * R', G * G', I * I' \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w;q*p'} (\mathbb{D}, \Sigma).$$

Since $(\sigma, w, \mathbb{D}, \Sigma) \models p * p'$, we know: there exist σ_1, σ_2, w_1, w_2, $\mathbb{D}_1$, $\mathbb{D}_2$, Σ_1 and Σ_2 such that

$$(\sigma_1, w_1, \mathbb{D}_1, \Sigma_1) \models p, \;\; (\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \models p', \;\; \sigma = \sigma_1 \uplus \sigma_2, \;\; w = w_1 + w_2, \;\; \mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2, \;\; \Sigma = \Sigma_1 \uplus \Sigma_2$$

From the premise, we know: $R, G, I \models (C, \sigma_1, (0, |C|)) \preceq_{\mathsf{height}(C);w_1;q} (\mathbb{D}_1, \Sigma_1)$.
By Lemma 45, we are done. □

Lemma 45. If

1. $R, G, I \models (C, \sigma_1, ws) \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma_1)$;

2. $\mathsf{Sta}(q, R * \mathsf{Id})$; $\mathsf{Sta}(p', (R')^+ * \mathsf{Id})$; $I \triangleright \{R, G\}$; $I' \triangleright \{R', G'\}$; $q \Rightarrow I * \mathbf{true}$; $p' \Rightarrow I' * \mathbf{true}$; $G^+ \Rightarrow G$;
3. $(\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \models p'$; $\sigma = \sigma_1 \uplus \sigma_2$; $\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$; $\Sigma = \Sigma_1 \uplus \Sigma_2$;

then $R * R', G * G', I * I' \models (C, \sigma, ws) \preceq_{\mathcal{H};w_1+w_2;q*p'} (\mathbb{D}, \Sigma)$.

Proof: By co-induction. From the premises, we know: $(\sigma_1, \Sigma_1) \models I * \mathbf{true}$ and $(\sigma_2, \Sigma_2) \models I' * \mathbf{true}$. Thus we know: $(\sigma, \Sigma) \models I * I' * \mathbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,

   from the first premise, we know: there exists σ'_1 such that $\sigma'' = \sigma'_1 \uplus \sigma_2 \uplus \sigma_F$, and one of the following holds:

   (a) there exist ws', w'_1, $\mathbb{C}'_1$ and Σ'_1 such that $(\mathbb{D}_1, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}'_1, \Sigma'_1 \uplus \Sigma_2 \uplus \Sigma_F)$, $((\sigma_1, \Sigma_1), (\sigma'_1, \Sigma'_1), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma'_1, ws') \preceq_{\mathcal{H};w'_1;q} (\mathbb{C}'_1, \Sigma'_1)$.
   Since $(\sigma_2, \Sigma_2) \models I' * \mathbf{true}$ and $I' \triangleright G'$, we know:

   $$((\sigma_2, \Sigma_2), (\sigma_2, \Sigma_2), \mathbf{true}) \models G' * \mathsf{True}.$$

   Since $G^+ \Rightarrow G$, we know:

   $$((\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2), (\sigma'_1 \uplus \sigma_2, \Sigma'_1 \uplus \Sigma_2), \mathbf{true}) \models (G * G')^+ * \mathsf{True}.$$

   Since $\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$, we know $\mathbb{D}_2 = \bullet$ and $\mathbb{D} = \mathbb{D}_1$. Let $\mathbb{D}' = \mathbb{C}'_1 \uplus \mathbb{D}_2 = \mathbb{C}'_1$.
   By the co-induction hypothesis, we know

   $$R * R', G * G', I * I' \models (C', \sigma'_1 \uplus \sigma_2, ws') \preceq_{\mathcal{H};w'_1+w_2;q*p'} (\mathbb{D}', \Sigma'_1 \uplus \Sigma_2).$$

   (b) there exists ws' such that $ws' <_{\mathcal{H}} ws$,
   $((\sigma_1, \Sigma_1), (\sigma'_1, \Sigma_1), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma'_1, ws') \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma_1)$.
   Since $(\sigma_2, \Sigma_2) \models I' * \mathbf{true}$ and $I' \triangleright G'$, we know:

   $$((\sigma_2, \Sigma_2), (\sigma_2, \Sigma_2), \mathbf{false}) \models G' * \mathsf{True}.$$

   Since $G^+ \Rightarrow G$, we know:

   $$((\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2), (\sigma'_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2), \mathbf{false}) \models (G * G')^+ * \mathsf{True}.$$

   By the co-induction hypothesis, we know

   $$R * R', G * G', I * I' \models (C', \sigma'_1 \uplus \sigma_2, ws') \preceq_{\mathcal{H};w_1+w_2;q*p'} (\mathbb{D}, \Sigma_1 \uplus \Sigma_2).$$

2. for any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$, the proof is similar to the previous case.

3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models (R * R')^+ * \mathsf{Id}$,

   since $I \triangleright R$, $I' \triangleright R'$, $(\sigma_1, \Sigma_1) \models I * \mathbf{true}$ and $(\sigma_2, \Sigma_2) \models I' * \mathbf{true}$, we know: there exist σ'_1, σ'_2, Σ'_1 and Σ'_2 such that $\sigma' = \sigma'_1 \uplus \sigma'_2$, $\Sigma' = \Sigma'_1 \uplus \Sigma'_2$,

   $$((\sigma_1, \Sigma_1), (\sigma'_1, \Sigma'_1), \mathbf{true}) \models R^+ * \mathsf{Id}, \quad ((\sigma_2, \Sigma_2), (\sigma'_2, \Sigma'_2), \mathbf{true}) \models (R')^+ * \mathsf{Id}$$

   From the first premise, we know there exist ws' and w'_1 such that

   $$R, G, I \models (C, \sigma'_1, ws') \preceq_{\mathcal{H};w'_1;q} (\mathbb{D}_1, \Sigma'_1).$$

   Since $(\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \models p'$ and $\mathsf{Sta}(p', (R')^+ * \mathsf{Id})$, we know: there exists w'_2 such that

   $$(\sigma'_2, w'_2, \mathbb{D}_2, \Sigma'_2) \models p'.$$

   By the co-induction hypothesis, we know:

$$R * R', G * G', I * I' \models (C, \sigma', ws') \preceq_{\mathcal{H};w_1'+w_2';q*p'} (\mathbb{D}, \Sigma').$$

4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models (R * R')^+ * \mathsf{Id}$,
   since $I \triangleright R$, $I' \triangleright R'$, $(\sigma_1, \Sigma_1) \models I * \mathbf{true}$ and $(\sigma_2, \Sigma_2) \models I' * \mathbf{true}$, we know: there exist σ_1', σ_2', Σ_1' and Σ_2' such that $\sigma' = \sigma_1' \uplus \sigma_2'$, $\Sigma' = \Sigma_1' \uplus \Sigma_2'$,

   $$((\sigma_1, \Sigma_1), (\sigma_1', \Sigma_1'), \mathbf{false}) \models R^+ * \mathsf{Id}, \quad ((\sigma_2, \Sigma_2), (\sigma_2', \Sigma_2'), \mathbf{false}) \models (R')^+ * \mathsf{Id}$$

   From the first premise, we know

   $$R, G, I \models (C, \sigma_1', ws) \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma_1').$$

   Since $(\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \models p'$ and $\mathsf{Sta}(p', (R')^+ * \mathsf{Id})$, we know:

   $$(\sigma_2', w_2, \mathbb{D}_2, \Sigma_2') \models p'.$$

   By the co-induction hypothesis, we know:

   $$R * R', G * G', I * I' \models (C, \sigma', ws) \preceq_{\mathcal{H};w_1+w_2;q*p'} (\mathbb{D}, \Sigma').$$

5. if $C = \mathbf{skip}$, then for any Σ_F, from the first premise we know one of the following holds:

   (a) there exist w_1', $\mathbb{C}_1'$ and Σ_1' such that $(\mathbb{D}_1, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}_1', \Sigma_1' \uplus \Sigma_2 \uplus \Sigma_F)$, $((\sigma_1, \Sigma_1), (\sigma_1, \Sigma_1'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma_1, w_1', \mathbb{C}_1', \Sigma_1') \models q$.
   Since $(\sigma_2, \Sigma_2) \models I' * \mathbf{true}$ and $I' \triangleright G'$, we know:

   $$((\sigma_2, \Sigma_2), (\sigma_2, \Sigma_2), \mathbf{true}) \models G' * \mathsf{True}.$$

   Since $G^+ \Rightarrow G$, we know:

   $$((\sigma_1 \uplus \sigma_2, \Sigma_1 \uplus \Sigma_2), (\sigma_1 \uplus \sigma_2, \Sigma_1' \uplus \Sigma_2), \mathbf{true}) \models (G * G')^+ * \mathsf{True}.$$

   Since $\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$, we know $\mathbb{D}_2 = \bullet$ and $\mathbb{D} = \mathbb{D}_1$. Thus $\mathbb{C}_1' \uplus \mathbb{D}_2 = \mathbb{C}_1'$.
   Since $(\sigma_1, w_1', \mathbb{C}_1', \Sigma_1') \models q$, we get:

   $$(\sigma, w_1' + w_2, \mathbb{C}_1' \uplus \mathbb{D}_2, \Sigma_1' \uplus \Sigma_2) \models q * p'.$$

   (b) there exists w_1' such that $ws = (w_1', 0)$ and $(\sigma_1, w_1 + w_1', \mathbb{D}_1, \Sigma_1) \models q$.
   Since $(\sigma_2, w_2, \mathbb{D}_2, \Sigma_2) \models p'$, we have

   $$(\sigma, w_1 + w_2 + w_1', \mathbb{D}, \Sigma) \models q * p'.$$

6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$,
   from the first premise, we know: $(\mathbb{D}_1, \Sigma_1 \uplus \Sigma_2 \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$. Thus $\mathbb{D}_2 = \bullet$ and $\mathbb{D} = \mathbb{D}_1$. Thus $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

The FR-CONJ rule.

Lemma 46 (FR-CONJ). If

1. $R, G, I \models \{p\}C\{q\}$;
2. $\mathsf{Sta}(\{p, q\}, R * \mathsf{Id})$; $\mathsf{Sta}(p', R^+ * \mathsf{Id})$; $\mathsf{Sta}(p', G * \mathsf{True})$; $I \triangleright \{R, G\}$; $p \vee q \Rightarrow I * \mathbf{true}$;

then $R, G, I \models \{p \circledcirc p'\}C\{q \circledcirc p'\}$.

Proof: We want to prove: for all σ, w, $\mathbb{D}$ and Σ, if $(\sigma, w, \mathbb{D}, \Sigma) \models p \circledcirc p'$, then

$$R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w;q \circledcirc p'} (\mathbb{D}, \Sigma).$$

Since $(\sigma, w, \mathbb{D}, \Sigma) \models p \circledcirc p'$, we know: there exist w_1, w_2, $\mathbb{D}_1$ and $\mathbb{D}_2$ such that

$$(\sigma, w_1, \mathbb{D}_1, \Sigma) \models p, \;\; (\sigma, w_2, \mathbb{D}_2, \Sigma) \models p', \;\; w = w_1 + w_2, \;\; \mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$$

From the premise, we know: $R, G, I \models (C, \sigma, (0, |C|)) \preceq_{\mathsf{height}(C);w_1;q} (\mathbb{D}_1, \Sigma)$.
By Lemma 47, we are done. □

Lemma 47. If

1. $R, G, I \models (C, \sigma, ws_1) \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma)$;
2. $\mathsf{Sta}(q, R * \mathsf{Id})$; $\mathsf{Sta}(p', R^+ * \mathsf{Id})$; $\mathsf{Sta}(p', G * \mathsf{True})$; $I \triangleright \{R, G\}$; $q \Rightarrow I * \mathbf{true}$;
3. $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$; $w = w_1 + w_2$; $\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$;

then $R, G, I \models (C, \sigma, ws_1) \preceq_{\mathcal{H};w;q \circledcirc p'} (\mathbb{D}, \Sigma)$.

Proof: By co-induction. From the premises, we know: $(\sigma, \Sigma) \models I * \mathbf{true}$.

1. for any σ_F, Σ_F, C' and σ'', if $(C, \sigma \uplus \sigma_F) \longrightarrow (C', \sigma'')$,
   from the first premise, we know: there exists σ' such that $\sigma'' = \sigma' \uplus \sigma_F$, and one of the following holds:
   (a) there exist ws_1', $\mathbb{C}_1'$ and Σ' such that $(\mathbb{D}_1, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}_1', \Sigma' \uplus \Sigma_F)$,
   $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws_1') \preceq_{\mathcal{H};w_1;q} (\mathbb{C}_1', \Sigma')$.
   Since $\mathsf{Sta}(p', G * \mathsf{True})$, we know
   $$\mathsf{Sta}(p', G^+ * \mathsf{True})$$
   Since $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$, we know there exists w_2' such that
   $$(\sigma', w_2', \mathbb{D}_2, \Sigma') \models p'$$
   Since $\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$, we know $\mathbb{D}_2 = \bullet$ and $\mathbb{D} = \mathbb{D}_1$. Let $\mathbb{D}' = \mathbb{C}_1' \uplus \mathbb{D}_2 = \mathbb{C}_1'$ and $w' = w_1 + w_2'$.
   By the co-induction hypothesis, we know
   $$R, G, I \models (C', \sigma', ws_1') \preceq_{\mathcal{H};w';q \circledcirc p'} (\mathbb{D}', \Sigma').$$
   (b) there exists ws_1' such that $ws_1' <_{\mathcal{H}} ws_1$,
   $((\sigma, \Sigma), (\sigma', \Sigma), \mathbf{false}) \models G^+ * \mathsf{True}$ and $R, G, I \models (C', \sigma', ws_1') \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma)$.
   Since $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$ and $\mathsf{Sta}(p', G * \mathsf{True})$, we know
   $$(\sigma', w_2, \mathbb{D}_2, \Sigma) \models p'$$
   By the co-induction hypothesis, we know
   $$R, G, I \models (C', \sigma', ws_1') \preceq_{\mathcal{H};w;q \circledcirc p'} (\mathbb{D}, \Sigma).$$

2. for any σ_F, Σ_F, e, C' and σ'', if $(C, \sigma \uplus \sigma_F) \xrightarrow{e} (C', \sigma'')$, the proof is similar to the previous case.

3. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{true}) \models R^+ * \mathsf{Id}$,
   from the first premise, we know there exists ws_1' such that

$$R, G, I \models (C, \sigma', ws_1') \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma').$$

   Since $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$ and $\mathsf{Sta}(p', R^+ * \mathsf{Id})$, we know: there exists w_2' such that

$$(\sigma', w_2', \mathbb{D}_2, \Sigma') \models p'.$$

   By the co-induction hypothesis, we know: let $w' = w_1 + w_2'$,

$$R, G, I \models (C, \sigma', ws_1') \preceq_{\mathcal{H};w';q \circledcirc p'} (\mathbb{D}, \Sigma').$$

4. for any σ' and Σ', if $((\sigma, \Sigma), (\sigma', \Sigma'), \mathbf{false}) \models R^+ * \mathsf{Id}$,
   from the first premise, we know

$$R, G, I \models (C, \sigma', ws_1) \preceq_{\mathcal{H};w_1;q} (\mathbb{D}_1, \Sigma').$$

   Since $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$ and $\mathsf{Sta}(p', R^+ * \mathsf{Id})$, we know:

$$(\sigma', w_2, \mathbb{D}_2, \Sigma') \models p'.$$

   By the co-induction hypothesis, we know:

$$R, G, I \models (C, \sigma', ws_1) \preceq_{\mathcal{H};w;q \circledcirc p'} (\mathbb{D}, \Sigma').$$

5. if $C = \mathbf{skip}$, then for any Σ_F, from the first premise we know one of the following holds:

   (a) there exist w_1', $\mathbb{C}_1'$ and Σ' such that $(\mathbb{D}_1, \Sigma \uplus \Sigma_F) \longrightarrow^+ (\mathbb{C}_1', \Sigma' \uplus \Sigma_F)$, $((\sigma, \Sigma), (\sigma, \Sigma'), \mathbf{true}) \models G^+ * \mathsf{True}$ and $(\sigma, w_1', \mathbb{C}_1', \Sigma') \models q$.
   Since $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$ and $\mathsf{Sta}(p', G * \mathsf{True})$, we know there exists w_2' such that

$$(\sigma, w_2', \mathbb{D}_2, \Sigma') \models p'$$

   Since $\mathbb{D} = \mathbb{D}_1 \uplus \mathbb{D}_2$, we know $\mathbb{D}_2 = \bullet$ and $\mathbb{D} = \mathbb{D}_1$. Thus $\mathbb{C}_1' \uplus \mathbb{D}_2 = \mathbb{C}_1'$. Thus we get:

$$(\sigma, w_1' + w_2', \mathbb{C}_1' \uplus \mathbb{D}_2, \Sigma') \models q \circledcirc p'.$$

   (b) there exists w_1' such that $ws_1 = (w_1', 0)$ and $(\sigma, w_1 + w_1', \mathbb{D}_1, \Sigma) \models q$.
   Since $(\sigma, w_2, \mathbb{D}_2, \Sigma) \models p'$, we have

$$(\sigma, w_1 + w_2 + w_1', \mathbb{D}, \Sigma) \models q \circledcirc p'.$$

6. for any σ_F and Σ_F, if $(C, \sigma \uplus \sigma_F) \longrightarrow \mathbf{abort}$,
   from the first premise, we know: $(\mathbb{D}_1, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$. Thus $\mathbb{D}_2 = \bullet$ and $\mathbb{D} = \mathbb{D}_1$. Thus $(\mathbb{D}, \Sigma \uplus \Sigma_F) \longrightarrow^+ \mathbf{abort}$.

Thus we are done. □

5.5 Derivation of WHILE-TERM Rule

Lemma 48 (WHILE-TERM Derivable). If

1. $R, G, I \vdash \{p \wedge B \wedge (E = \alpha)\} C \{p \wedge (E < \alpha)\}$;
2. $p \wedge B \Rightarrow E > 0$;
3. $p \Rightarrow ((B = B) \wedge (E = E)) * I$;
4. $G^+ \Rightarrow G$;
5. α is a fresh logical variable;

then $R, G, I \vdash \{\lfloor p \rfloor_{\mathsf{w}}\}\textbf{while}\ (B)\ C\{\lfloor p \rfloor_{\mathsf{w}} \wedge \neg B\}$.

Proof: Take a fresh logical variable β and by applying the CONSEQ rule to the premise 1, we get:

$$R, G, I \vdash \{\exists \beta.\ p \wedge (E = \beta) \wedge B \wedge (E = \alpha)\} C \{\exists \beta.\ p \wedge (E = \beta) \wedge (E < \alpha)\} \tag{5.178}$$

From $p \wedge B \Rightarrow E > 0$, we know

$$p \wedge B \wedge (E = \alpha) \ \Rightarrow\ \alpha > 0 \tag{5.179}$$

Since $G^+ \Rightarrow G$, $\mathsf{Sta}(\mathsf{wf}(\alpha) \wedge \mathsf{emp}, \mathsf{Emp} * \mathsf{Id})$, $\mathsf{emp} \triangleright \mathsf{Emp}$ and $(\mathsf{wf}(\alpha) \wedge \mathsf{emp}) \Rightarrow \mathsf{emp} * \textbf{true}$, we can apply the FRAME rule to (5.178) and get

$$R, G, I \vdash \{(\exists \beta.\ p \wedge (E = \beta) \wedge B \wedge (E = \alpha)) * (\mathsf{wf}(\alpha) \wedge \mathsf{emp})\} C \{(\exists \beta.\ p \wedge (E = \beta) \wedge (E < \alpha)) * (\mathsf{wf}(\alpha) \wedge \mathsf{emp})\} \tag{5.180}$$

We reduce (5.180) as follows:

$$R, G, I \vdash \{\exists \beta.\ (p \wedge (E = \beta)) * (\mathsf{wf}(\alpha) \wedge \mathsf{emp}) \wedge B \wedge (E = \alpha)\} C \{\exists \beta.\ (p \wedge (E = \beta)) * (\mathsf{wf}(\alpha) \wedge \mathsf{emp}) \wedge (E < \alpha)\} \tag{5.181}$$

$$R, G, I \vdash \{\exists \beta.\ (p \wedge (E = \beta)) * (\mathsf{wf}(\beta) \wedge \mathsf{emp})) \wedge B \wedge (E = \alpha)\} C \{\exists \beta.\ (p \wedge (E = \beta)) * (\mathsf{wf}(\beta{+}1) \wedge \mathsf{emp})) \wedge (E < \alpha)\} \tag{5.182}$$

Since $(\mathsf{wf}(\beta + 1) \wedge \mathsf{emp}) \Rightarrow (\mathsf{wf}(\beta) \wedge \mathsf{emp}) * (\mathsf{wf}(1) \wedge \mathsf{emp})$, we let

$$p_0 \ \stackrel{\text{def}}{=}\ (\exists \beta.\ (p \wedge (E = \beta)) * (\mathsf{wf}(\beta) \wedge \mathsf{emp})) \tag{5.183}$$

then (5.182) can be written as:

$$R, G, I \vdash \{p_0 \wedge B \wedge (E = \alpha)\} C \{(p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp})) \wedge (E < \alpha)\} \tag{5.184}$$

By the EXISTS rule and α is not free in R, G and I, we get:

$$R, G, I \vdash \{\exists \alpha.\ p_0 \wedge B \wedge (E = \alpha)\} C \{\exists \alpha.\ (p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp})) \wedge (E < \alpha)\} \tag{5.185}$$

Since α is not free in p, B and E, we know

$$(p_0 \wedge B) \ \Rightarrow\ (\exists \alpha.\ p_0 \wedge B \wedge (E = \alpha)) \tag{5.186}$$

and

$$(\exists \alpha.\ (p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp})) \wedge (E < \alpha)) \ \Rightarrow\ (p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp})) \tag{5.187}$$

Thus by applying CONSEQ rule to (5.185), we get:

$$R, G, I \vdash \{p_0 \wedge B\} C \{p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp})\} \tag{5.188}$$

From $p \Rightarrow (B = B) * I$ and $p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp}) \wedge B \Rightarrow (p_0 \wedge B) * (\mathsf{wf}(1) \wedge \mathsf{emp})$, by applying the WHILE rule and the HIDE-W rule, we get:

$$R, G, I \vdash \{\lfloor p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp}) \rfloor_{\mathsf{w}}\}\textbf{while}\ (B)\ C\{\lfloor p_0 * (\mathsf{wf}(1) \wedge \mathsf{emp}) \rfloor_{\mathsf{w}} \wedge \neg B\} \tag{5.189}$$

It can be reduced to:

$$R, G, I \vdash \{\exists \beta.\ \lfloor p \rfloor_{\mathtt{w}} \wedge (E = \beta)\}\textbf{while}\ (B)\ C\{\exists \beta.\ \lfloor p \rfloor_{\mathtt{w}} \wedge (E = \beta) \wedge \neg B\} \tag{5.190}$$

Since $p \Rightarrow (E = E) * I$, we know

$$R, G, I \vdash \{\lfloor p \rfloor_{\mathtt{w}}\}\textbf{while}\ (B)\ C\{\lfloor p \rfloor_{\mathtt{w}} \wedge \neg B\} \tag{5.191}$$

Thus we are done. □

References

[1] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE'04*.

[2] Xinyu Feng. Local rely-guarantee reasoning. In *POPL'09*.

[3] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*.

[4] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[5] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *LICS*, pages 124–133, 2013.

[6] Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470, 2013.

[7] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[8] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC'96*.

[9] William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. In *PPoPP*, pages 147–156, 2006.

[10] Ketil Stølen. A method for the development of totally correct shared-state parallel programs. In *CONCUR*, pages 510–525, 1991.

[11] Aaron Turon and Mitchell Wand. A separation logic for refining concurrent objects. In *POPL'11*.

[12] Viktor Vafeiadis. Modular fine-grained concurrency verification. Thesis.

[13] Viktor Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, 2011.

A Compositional Semantics for Verified Separate Compilation and Linking

Tahina Ramananandro Zhong Shao Shu-Chun Weng Jérémie Koenig Yuchen Fu[1]

Yale University [1]Massachusetts Institute of Technology

Abstract

Recent ground-breaking efforts such as CompCert have made a convincing case that mechanized verification of the compiler correctness for realistic C programs is both viable and practical. Unfortunately, existing verified compilers can only handle whole programs—this severely limits their applicability and prevents the linking of verified C programs with verified external libraries. In this paper, we present a novel compositional semantics for reasoning about open modules and for supporting verified separate compilation and linking. More specifically, we replace external function calls with explicit events in the behavioral semantics. We then develop a verified linking operator that makes lazy substitutions on (potentially reacting) behaviors by replacing each external function call event with a behavior simulating the requested function. Finally, we show how our new semantics can be applied to build a refinement infrastructure that supports both vertical composition and horizontal composition.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors—Compilers; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods

Keywords Compositional Semantics; Vertical Composition; Horizontal Composition; Verified Compilation and Linking.

1. Introduction

Compiler verification has long been considered as a theoretically deep and practically important research subject. It addresses the very question of program equivalence (or simulation), a primary reason that we need to define formal semantics for programming languages. It is important for practical software developers since compiler bugs can lead to the silent generation of incorrect programs, which could lead to unexpected crashes and security holes.

Recent work on CompCert [12, 11] has shown that mechanized verification of the compiler correctness for C is both viable and practical, and the resulting compiler is indeed empirically much more reliable than traditional (unverified) ones [22]. The success of CompCert can be partly attributed to its uses of simple (small-step and/or big-step) operational semantics [14], a shared behavioral specification language (capable of describing terminating, stuck, silently diverging, and reacting behaviors), and a unified C memory model [13] for all of its compiler intermediate languages. The simplicity of the CompCert semantics made it possible and practical to mechanically verify the correctness of many compilation phases under a reasonable amount of effort.

One important weakness of CompCert is that it can only handle whole programs. This severely limits its applicability. A computer program is often not just a single piece of code written and compiled at once, but is instead obtained by compiling and linking different *modules*, or *compilation units*, that can be originally written in different programming languages, independently of each other. From the compilation point of view, the final program is obtained by linking different *object files*, each of which is either written directly or obtained by compiling a source compilation unit. Different compilers can be used for different modules.

From the program-verification point of view, a computer program is almost never verified as a whole, but for each compilation unit, its source code (or object file, if written directly) is verified independently from the implementation of the other modules. Without support for separate compilation and linking, verified C programs, even if correctly compiled by CompCert, cannot be linked with verified external libraries.

An open problem for supporting verified separate compilation and linking is to find a simple *compositional* semantics for open modules and to specify and reason about such semantic behaviors in a language-independent way. Following Hur *et al* [9, 10], we want to achieve compositionality in the two dimensions:

- *vertical composition* corresponds to successive compilation passes on a given compilation unit. Each compilation pass can be an optimization to make a program more efficient while staying at the same representation level, or a compilation phase from one intermediate representation to another: how to define compositional semantics of intermediate programs in a language-independent format so that we can show that each compilation pass does not introduce unwanted behaviors?
- *horizontal composition* corresponds to the linking of different modules at the same level (i.e. at the level of object files, or at the same intermediate level). It corresponds to the notion of *program composition*: local reasoning shall allow studying the behavior of program components when placed in an abstractly specified context. But conversely, when linking them together, compilation units will play the role of contexts for other modules. More generally, this notion becomes symmetric when they can mutually call functions in each other.

In this paper, we present a novel compositional semantics (for open modules) that supports both vertical composition and hori-

CPP '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3296-5/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693167

zontal composition for C-like languages. Traditionally, operational semantics focuses on reasoning about the behaviors of a whole program. This partly explains why CompCert does not handle open modules. A significant attempt toward developing compositional semantics has been denotational semantics, and the underlying domain theory has led to a wide body of research; however, denotational models become difficult to extend as we add more language features and they are harder to mechanize in a proof assistant.

Our paper makes the following contributions:

- We develop a *compositional semantics* (denoted as $[\![\cdot]\!]_{\mathsf{comp}}$, see Sec. 4) to help reason about open modules. Our key idea is to model external function calls in a similar way as how compositional semantics for concurrent languages [4] models environmental transitions. The behavior of a call to an external function f is modeled as an event $\texttt{Extcall}(f, m, m')$, with m and m' denoting memory states before and after the call. A function body that makes n consecutive external calls can be modeled as a sequence of event traces of the form $\texttt{Extcall}(f_1, m_1, m_1') :: \texttt{Extcall}(f_2, m_2, m_2') :: \ldots :: \texttt{Extcall}(f_n, m_n, m_n')$, with the assumption that segments between two external call events, e.g., (m_1', m_2) and (m_{n-1}', m_n), are transitions made by the function body itself. We show how to extend the CompCert-style behavioral semantics with these new external call events and how to use a shared behavioral specification language (as in CompCert) to support vertical compositionality.
- We develop a *linking* operator directly at the semantic level (denoted as $\bowtie$, see Sec. 5), based on a resolution operator which makes a lazy substitution on behaviors by replacing each external function call event with a behavior simulating the requested function. We show that applying the linking operator to the compositional semantic objects (ψ_1 and ψ_2 for open modules $\mathfrak{u}_1$ and $\mathfrak{u}_2$) will yield the same compositional semantic object ($\psi_1 \bowtie \psi_2$) for the linked module ($\mathfrak{u}_1 \uplus \mathfrak{u}_2$). Since linking is directly done on semantic objects, our approach can also be applied to components compiled from different source languages; for example, a module $\mathfrak{u}_\alpha$ (in language A) can be compiled by compiler C_α and linked with another module $\mathfrak{u}_\beta$ compiled by compiler C_β, yielding a resulting binary with the semantic object $[\![C_\alpha(\mathfrak{u}_\alpha)]\!]_{\mathsf{comp}} \bowtie [\![C_\beta(\mathfrak{u}_\beta)]\!]_{\mathsf{comp}}$.
- Thanks to this new compositional semantics and semantic linking, we develop a refinement infrastructure (denoted as $\sqsubseteq$, see Sec. 6) that unifies program verification and verified separate compilation: each verification step, as well as each compilation step, is actually a refinement step. The transitivity property of our refinement relation implies vertical composition; and the congruence property (a.k.a. monotonicity, see Theorem 2) implies horizontal composition.
- Unlike the CompCert whole program semantics, which does not expose memory states in its event traces, compositional semantics for open modules may make part of the memory state observable (e.g., as in an external call event $\texttt{Extcall}(f, m, m')$). This creates challenges for verifying compilation phases that alter memory states. We introduce α-refinement (denoted as $\sqsubseteq_\alpha$, see Sec. 7), a generalization of $\sqsubseteq$ with a bijection α between the source and the target memory states. We show how α-refinement can be used to verify the correctness of the memory-changing phases in CompCert, and we have successfully reimplemented (and verified in Coq) the Clight-to-Cminor phase—the CompCert pass that uses the most sophisticated memory injection relation—using α-like memory bijection.

All our proofs have been carried out in Coq [20] and can be found at the companion web site [18]. The implementation includes the generic compositional semantics and linking framework, an instantiation of the framework for the common subexpression elimination pass, and a new implementation of the CompCert memory model with block tags and the Clight-to-Cminor compilation phase using memory bijection.

2. Preliminary: small-step and big-step semantics

In this section, we define the general notion of small-step semantics, or transition systems, and explain how to automatically construct big-step semantics based on them. Throughout the paper, when we define a small-step semantics, we always construct the corresponding big-step semantics based on this section.

Small-step semantics illustrates how to execute programs with minimal steps. Big-step semantics gives us the meaning of programs as a whole. When studying the meaning of a program, we focus not only on whether it terminates or diverges, but also on its interaction with the outside environment through *events* like input and output, network communications, etc. We borrow all these definitions from the CompCert verified compiler [11].

Before diving into semantics, we first go through some notations on sets, (finite) lists, and (infinite) streams. We use $X^?$ to denote the set of all subsets of X with 0 or 1 element. For any subset $Y \subseteq X^?$, we liberally write $x \in Y$ instead of $\{x\} \in Y$. The standard notation for power set $\mathcal{P}(X)$ is also used.

For any set X, X^* denotes the set of finite lists of elements of X. Such lists can be either empty (ε) or nonempty ($x :: l$). For two lists $l_1, l_2 \in X^*$, $l_1 +\!\!+ l_2$ is their concatenation. X^∞ denotes the set of infinite streams of X, which are defined coinductively such that all elements are of the form $x :::: \mathfrak{l}$ where $x \in X$ and $\mathfrak{l} \in X^\infty$. The coinductive definition allows (actually, requires) streams to be infinite, in contrast to lists, which are defined inductively and must be finite. Prepending a list l of X in front of a stream $\mathfrak{l}$ of X is written $l +\!\!+\!\!+ \mathfrak{l}$. We write $\mathfrak{l}_1 \sim \mathfrak{l}_2$ meaning two streams are bisimilar (coinductively, $\exists x, \forall i = 1, 2, \exists \mathfrak{l}_i', \mathfrak{l}_i = x :::: \mathfrak{l}_i'$ and $\mathfrak{l}_1' \sim \mathfrak{l}_2'$).

Definition 1 (Small-step semantics)**.** *A* small-step semantics *(or a* transition system*) is a tuple* $\mathfrak{S} = (\mathcal{E}, \mathcal{S}, \rightarrow, \mathcal{R}, \mathcal{F})$ *where:*

- $\mathcal{E}$ *is the set of* events.
- $\mathcal{S}$ *is the set of* configurations *(or* states*).*
- $(\rightarrow) \subseteq \mathcal{S} \times \mathcal{E}^? \times \mathcal{S}$ *is the* transition relation*, usually written in infix forms* $s \xrightarrow{e} s'$ *and* $s \rightarrow s'$*. We say that* s *makes one step (or transition) to* s'*, producing an event* e *(if any). A step producing no event is* silent*.*
- $\mathcal{R}$ *is the set of* results*.*
- $\mathcal{F} \subseteq (\mathcal{S} \times \mathcal{R})$ *is a relation associating* final states *with* results*. A configuration* s *is said to be final with result* r *if, and only if,* $(s, r) \in \mathcal{F}$*.*

The transition relation may be *nondeterministic*: for a given configuration s, there can be several possible configurations s' such that $s \rightarrow s'$ (or $s \xrightarrow{e} s'$ for some event e).

Then, a configuration s can make several transitions to s' producing a finite list σ of events in $\mathcal{E}$, which we write $s \xrightarrow{\sigma}{}^* s'$ (or $s \xrightarrow{\sigma}{}^+ s'$ if there is at least one step) and define as the reflexive-transitive (resp. transitive) closure of the transition step relation:

$$\frac{s \rightarrow s'}{s \xrightarrow{\varepsilon}{}^+ s'} \qquad \frac{s \xrightarrow{e} s'}{s \xrightarrow{e::\varepsilon}{}^+ s'} \qquad \frac{s \xrightarrow{\sigma_1}{}^+ s_1 \quad s_1 \xrightarrow{\sigma_2}{}^+ s_2}{s \xrightarrow{\sigma_1 +\!\!+ \sigma_2}{}^+ s_2} \qquad \frac{}{s \xrightarrow{\varepsilon}{}^* s} \qquad \frac{s \xrightarrow{\sigma}{}^+ s'}{s \xrightarrow{\sigma}{}^* s'}$$

We can then define the *behavior* of a transition system from an initial state $s_0 \in \mathcal{S}$.

- It can perform finitely many transition steps to some final configuration s' such that $(s', r') \in \mathcal{F}$ for some r'. In this case, we say that it is a *terminating* behavior. For such a behavior, we

record the result r' and its *trace* of events, a finite list, produced to go from s_0 to s'.

- It can perform finitely many transition steps to some non-final configuration s' but from which no step is possible. In this case, we say that it is a *going-wrong* (or *stuck*) behavior, for which we record the trace produced from s_0 to s'. In practice, s' corresponds to a configuration requesting an invalid operation such as out-of-bounds array access or division by zero.
- It can perform infinitely many transition steps. For such cases, we need to distinguish whether a finite or infinite list of events is produced during these transition steps. (1) In the finite event case, finitely many steps are performed to some state s' from which infinitely many silent transitions are performed. (2) In the infinite event case, starting from any state, a non-silent transition can always be reached within finitely many steps; we record the trace as an infinite stream of events.

The bullets above are formally defined as follows.

Definition 2 (Behaviors)**.** *Given a set of events $\mathcal{E}$ and a set of results $\mathcal{R}$, we define the set of behaviors $\mathcal{B}$ as follows:*

b	$\in$	$\mathcal{B}$		*Behavior*
	::=	$\sigma\downarrow(r)$	$(\sigma \in \mathcal{E}^*, r \in \mathcal{R})$	*Terminating behavior*
	|	$\sigma\lightning$	$(\sigma \in \mathcal{E}^*)$	*Going-wrong behavior*
	|	$\sigma\nearrow$	$(\sigma \in \mathcal{E}^*)$	*Diverging behavior*
			(finitely many events, then silently diverges)	
	|	$\varsigma\Lightarrow$	$(\varsigma \in \mathcal{E}^\infty)$	*Reacting behavior*
			(diverging with infinitely many events)	

The concatenation of an event e (resp. of an event list σ) and a behavior b is written $e \cdot b$ (resp. $\sigma \bullet b$) and defined in a straightforward way:

$$\begin{array}{rclcrcl} e\cdot(\sigma\downarrow(r)) & = & (e::\sigma)\downarrow(r) & \qquad & \varepsilon\bullet b & = & b \\ e\cdot(\sigma\lightning) & = & (e::\sigma)\lightning & & (e::\sigma)\bullet b & = & e\cdot(\sigma\bullet b) \\ e\cdot(\sigma\nearrow) & = & (e::\sigma)\nearrow & & & & \\ e\cdot(\varsigma\Lightarrow) & = & (e::::\varsigma)\Lightarrow & & & & \end{array}$$

Definition 3 (Stuck, silently diverging, reacting states)**.** *Given a small-step semantics $\Xi = (\mathcal{E}, \mathcal{S}, \rightarrow, \mathcal{R}, \mathcal{F})$, a configuration s is:*

- stuck *(written $s\lightning$) if, and only if, there is no s' (resp. and there is no $e \in \mathcal{E}$) such that $s \rightarrow s'$ (resp. $s \xrightarrow{e} s'$)*
- silently diverging *(written $s\nearrow$) if, and only if, coinductively, there is a configuration s' such that $s \rightarrow s'$ and $s'\nearrow$.*
- reacting *with the infinite event stream ς (written $s\Lightarrow\varsigma$) if, and only if, coinductively, there is a* nonempty *finite event list σ and a configuration s' such that $s \xrightarrow{\sigma}{}^{+} s'$, and an infinite event stream ς' such that $s'\Lightarrow\varsigma'$ and $\varsigma \sim \sigma \mathbin{⧺} \varsigma'$.*

If the transition relation is nondeterministic, the transition system may have several behaviors from a single initial state. We want to describe the set of all the possible behaviors of the transition system from a given configuration.

Definition 4 (Big-step semantics)**.** *Given a small-step semantics $\Xi = (\mathcal{E}, \mathcal{S}, \rightarrow, \mathcal{R}, \mathcal{F})$, the* big-step *semantics $(\!|\Xi|\!)$ of Ξ is a function from $\mathcal{S}$ to $\mathcal{P}(\mathcal{B})$ such that, for each configuration s_0, $(\!|\Xi|\!)(s_0)$ is the set of all possible behaviors from s_0, defined as follows:*

$$\begin{array}{rcl} (\!|\Xi|\!)(s_0) & = & \{\sigma\downarrow(r) : s_0 \xrightarrow{\sigma}{}^{*} s \wedge (s,r) \in \mathcal{F}\} \\ & \cup & \{\sigma\lightning : s_0 \xrightarrow{\sigma}{}^{*} s \ \wedge \ s\lightning\} \\ & \cup & \{\sigma\nearrow : s_0 \xrightarrow{\sigma}{}^{*} s \ \wedge \ s\nearrow\} \\ & \cup & \{\varsigma\Lightarrow : s_0 \Lightarrow \varsigma\} \end{array}$$

Below are some C examples showing the use of behaviors. The command `printf('a');`, printing the character "a" on screen, produces an observable event `OUT(a)`. The results are `int` values.

```
int main () { printf('a'); return 2; }
```
has the behavior `OUT(a)` :: $\varepsilon\downarrow(2)$.

```
int main () { printf('a'); 3/0; return 4;}
```
has the behavior `OUT(a)` :: $\varepsilon\lightning$.

```
int main () { printf('a'); while (1) {} }
```
has the behavior `OUT(a)` :: $\varepsilon\nearrow$.

```
int main () { while (1) printf('b'); }
```
has the behavior `OUT(b)` :::: ... :::: `OUT(b)` :::: ... $\Lightarrow$

Lemma 1. *For any configuration $s_0 \in \mathcal{S}$, the transition system has at least one behavior from s_0: $(\!|\Xi|\!)(s_0) \neq \varnothing$.*

Proof. Done in CompCert [11]. Requires the excluded middle to distinguish whether the program has finite or infinite sequence of steps, and an axiom of constructive indefinite description to construct the infinite event sequence in the reacting case. □

3. Starting point: a language with function calls

Our work studies a semantic notion of linking two compilation units at the level of their behaviors, independently of the languages in which they are defined. We first show how to derive a set of behaviors for an open module from a language with function calls.

In this section, we first describe our starting point, the semantics of a language with function calls. For now, we consider only whole programs. Then we will show in Sec. 4 how to make its semantics compositional and suitable for open modules.

Our starting point language makes a memory state evolve throughout the whole program execution across function calls, and a local state (e.g. local variables) evolve within each function call. When a function returns, we consider that its result is the new memory state obtained at the end of the execution of the function, just before it hands over to its caller. Our Coq development also features argument passing and return value, but for the sake of presentation, we do not mention them here. See Sec. 6.5 for more details about our Coq implementation.

The key point of the semantics of our language is that the local state of a function call cannot be changed by other function calls: when a function is called, the local state of the caller is "frozen" until the callee returns.

In this section, we consider the semantics of a *whole program*, which does not contain external function calls. A program consists of several functions. We are interested in the behaviors of each function in the program for all memory states under which the function is called.

A program is modeled as a partial function from function names to code. Let p be a program and f be a function name, $p(f)$ is then the body of f. When f is called under a memory state m, an initial local state $\mathsf{Init}(p(f), m)$ is first created from the code $p(f)$. The local state and the memory state evolve together by performing *local transition steps* which can produce some events. Eventually, the local state may correspond to a *return state*, meaning that the execution of the function has reached termination. The memory state is the result of the function call.

But a local state l can also correspond to *calling* some other function f': in that case, l is first saved into a *continuation frame* $k = \mathsf{Backup}(m, l)$ that is put on top of a *continuation stack*, then the function f' is called and run. If the execution of this callee reaches a return state, with a new memory state m', then the execution goes back to the caller by retrieving, from the stack, the frame k and constructing a new local state $\mathsf{Restore}(m', k)$.

So, to obtain the behaviors of a function f under a memory state m, we just have to big-step such a small-step semantics. Note that when an execution reaches a configuration where the local state is a return state and the stack is empty, it means that the function is done executing and there is no caller function to return to, hence it

is the final configuration for a function execution. The result of the execution is the memory state of such a configuration.

Definition 5 (Language with function calls)**.** *A* language with function calls *is a tuple:*

$$\mathfrak{L} = (\mathbf{F}, \mathbf{C}, \mathbf{MS}, \mathbf{LS}, \mathsf{Init}, \mathsf{Kind}, \mathbf{E}, \rightarrow, \mathbf{K}, \mathsf{Backup}, \mathsf{Restore})$$

- **F** *is the set of* function names.
- **C** *is the set of pieces of* code *corresponding to the bodies of functions (i.e., the* syntax *of the language).*
- **MS** *is the set of* memory states.
- **LS** *is the set of* local states.
- $\mathsf{Init} : (\mathbf{C} \times \mathbf{MS}) \longrightarrow \mathbf{LS}$ *is a total function that gives the initial local state when starting to execute a function body.*
- Kind *is a total function such that, for any local state* $l \in \mathbf{LS}$*,* $\mathsf{Kind}(l)$ *may be either:*
  - $\texttt{Call}(f)$ *to say that* l *corresponds to calling a function* $f \in \mathbf{F}$*. Then we define* $\mathbf{LS}_{\texttt{Call}} = \{l : \exists f, \mathsf{Kind}(l) = \texttt{Call}(f)\}$.
  - `Return` *to say that* l *is a return state.*
  - `Normal`*: none of the above. Then, we define* $\mathbf{LS}_{\texttt{Normal}} = \{l : \mathsf{Kind}(l) = \texttt{Normal}\}$.
- **E** *is the set of* events*.*
- $(\rightarrow) \subseteq ((\mathbf{MS} \times \mathbf{LS}_{\texttt{Normal}}) \times \mathbf{E}^? \times (\mathbf{MS} \times \mathbf{LS}))$ *is the* internal step *relation, usually written in infix forms* $(m, l) \xrightarrow{e} (m', l')$ *and* $(m, l) \rightarrow (m', l')$*.*
- **K** *is the set of* continuation stack frames*.*
- $\mathsf{Backup} : (\mathbf{MS} \times \mathbf{LS}_{\texttt{Call}}) \longrightarrow \mathbf{K}$ *is a total function that* saves *the current local state into a stack frame upon function call.*
- $\mathsf{Restore} : (\mathbf{MS} \times \mathbf{K}) \longrightarrow \mathbf{LS}$ *is a total function that* restores *a new local state from a stack frame upon callee return.*

Definition 6. *Let* $\mathfrak{L}$ *be a language with function calls. A* program *is a partial function from function names to code.*

Definition 7 (Procedural semantics)**.** *Let* $\mathfrak{L}$ *be a language with function calls and* p *be a program in* $\mathfrak{L}$*, the* procedural small-step semantics $\mathsf{Proc}[\mathfrak{L}, p]$ *is defined as follows:*

- *The set of events is* **E***.*
- *The set of configurations is* $\mathbf{MS} \times \mathbf{LS} \times \mathbf{K}^*$*.*
- *The transition relation* $(\mathfrak{L}, p) \vdash \cdot \rightarrow \cdot$ *is defined as follows:*

$$\frac{\mathsf{Kind}(l) = \texttt{Normal} \quad (m, l) \rightarrow (m', l')}{(\mathfrak{L}, p) \vdash (m, l, \kappa) \rightarrow (m', l', \kappa)}$$

$$\frac{\mathsf{Kind}(l) = \texttt{Normal} \quad (m, l) \xrightarrow{e} (m', l') \quad e \in \mathbf{E}}{(\mathfrak{L}, p) \vdash (m, l, \kappa) \xrightarrow{e} (m', l', \kappa)}$$

$$\frac{\mathsf{Kind}(l) = \texttt{Call}(f) \quad p(f) = c \quad l' = \mathsf{Init}(c, m) \quad k = \mathsf{Backup}(m, l)}{(\mathfrak{L}, p) \vdash (m, l, \kappa) \rightarrow (m, l', k :: \kappa)}$$

$$\frac{\mathsf{Kind}(l) = \texttt{Return} \quad l' = \mathsf{Restore}(m, k)}{(\mathfrak{L}, p) \vdash (m, l, k :: \kappa) \rightarrow (m, l', \kappa)}$$

- *The set of results is* **MS***.*
- *The final configurations with result* m *are the configurations* (m, l, ε) *where* $\mathsf{Kind}(l) = \texttt{Return}$*.*

Let **B** *be the set of behaviors on events* **E***. The* procedural big-step semantics *of* p *is the function* $[\![p]\!] : \mathrm{dom}(p) \longrightarrow \mathbf{MS} \longrightarrow \mathcal{P}(\mathbf{B})$ *obtained from big-stepping the procedural small-step semantics:*

$$[\![p]\!](f)(m) = (\!|\mathsf{Proc}[\mathfrak{L}, p]|\!)(m, \mathsf{Init}(p(f), m), \varepsilon)$$

Note that a function call is only triggered and the function name resolved when Kind returns $\texttt{Call}(f)$, which depends solely on local states. It does not matter how the calling request gets put into the local state. Whether it originates from the code, that is, a direct call, or prepared by the caller in the memory state only to be moved to the local state now, which indicates an indirect call, the procedural semantics handles them the same. This means that our setting transparently handles C-style higher-order function pointers, without having to provide a special case for them.

4. Compositional semantics

The procedural semantics given so far can only describe the behaviors of a *closed* program p. What if p calls a function outside of its domain? By definition, the execution goes wrong. As such, the procedural semantics alone is not compositional, and it is not enough to describe the behaviors of *open modules* (or *compilation units*).

In this section, we are going to make our procedural semantics compositional by extending it with a rule to handle external function calls, i.e. calls to functions that are not defined in the module.

This compositional semantics represents external function calls as events. We will later link two compilation units at the behavior level by replacing each external function call event with the behaviors of the callee (see Sec. 5).

The key idea of our compositional semantics is not to get stuck whenever a module calls an external function; instead, it produces a new form of event to record the external function call. This is consistent with the idea that events represent the interaction of a compilation unit with the outside environment: for an open module, external functions remain part of the outside environment until their implementation is provided by linking. These external call events are the minimal amount of syntax necessary to model external function calls at the level of behaviors.

For each external function call event, we record (1) the function name; (2) the memory state before the call, because the external call shall depend on the memory state under which it will be called; and (3) the memory state after function call. The external function may change the memory state arbitrarily, which the caller cannot control; but the behavior of the caller depends on how the callee changed the memory state.

For regular *input*, CompCert produces an ordinary event containing the value read from the environment. CompCert cannot predict the value, so it provides a behavior for every possible input. Which behavior appears at runtime will depend on the actual value read. More precisely, when a configuration requests an input, it must resort to nondeterminism and provide transition steps producing events for all possible values. Letting the event carry the value makes it possible to have different follow-up events or even termination status depending on the actual value read. For instance, in C, the command `scanf("%d", &i)` reads a value j from the keyboard and stores it into the variable `i`. CompCert models it as follows: for every integer j, there is a transition producing the event $\texttt{IN}(j)$ asserting that j is the value read. The following C code:

```
int main () {
  int i=0;
  scanf("%d", &i);
  printf("%d", (i%2));
  return 0;
}
```

will produce the set of behaviors [1]:

$$\{\texttt{IN}(j) :: \texttt{OUT}(j \bmod 2) :: \varepsilon \downarrow (0) : j \in [\mathsf{INT\_MIN}, \mathsf{INT\_MAX}]\}$$

We apply the same technique on external function calls. As the caller cannot predict how the callee will modify the memory state, the new memory state upon return of the function call is considered

[1] INT_MIN and INT_MAX are the least and the greatest values of type `int`.

as an external input from the environment. This is why an external function call event stores both the new and old memory states.

Then, when an external function f is called under some memory state m_1, for any possible memory state m_2 representing the memory state after returning from the external function, the compositional semantics will allow a transition (see rule EXTCALL in Def. 10 below) to produce the external function call event $\texttt{Extcall}(f, m_1, m_2)$. Consequently, the caller will be able to provide a behavior for each possible memory state m_2.

This leads us to extending the events and transition rules.

Definition 8 (Extended events)**.** *Let $\mathfrak{L}$ be a language with function calls. We write $\mathbb{E}$ for the set of* extended events*, defined as follows:*

$\mathfrak{e}$	$\in$	$\mathbb{E}$		*Extended event*
	::=	e	$(e \in \mathbf{E})$	*Regular event*
	|	$\texttt{Extcall}(f, m_1, m_2)$	$(f \in \mathbf{F},$ $m_1, m_2 \in \mathbf{MS})$	*External function call*

In $\texttt{Extcall}(f, m_1, m_2)$, m_1 and m_2 are the memory states before and after the call, respectively. We write $\mathbb{B}$ for the set of behaviors on events $\mathbb{E}$, which we call extended behaviors*.*

If $F \subseteq \mathbf{F}$ is a set of function names, then we write $\mathbb{E}_F$ for the set of extended events where all external function call events $\texttt{Extcall}(f, m_1, m_2)$ have $f \in F$, and $\mathbb{B}_F$ the set of behaviors on such events.

Definition 9 (Module or compilation unit)**.** *Let $\mathfrak{L}$ be a language with function calls. A* module*, or* compilation unit*, is a partial function from function names to code*[2]*.*

Definition 10 (Compositional semantics)**.** *The* compositional small-step semantics $\mathsf{Comp}[\mathfrak{L}, \mathfrak{u}]$ *of a compilation unit $\mathfrak{u}$ is the small-step semantics defined as follows:*

- *The set of events is $\mathbb{E}$.*
- *The set of configurations is $\mathbf{MS} \times \mathbf{LS} \times \mathbf{K}^*$ (as in the procedural small-step semantics).*
- *The transition relation $(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} \cdot \to \cdot$ is defined as follows:*

$$\frac{(\mathfrak{L}, \mathfrak{u}) \vdash (m, l, \kappa) \xrightarrow{e} (m', l', \kappa') \quad e \in \mathbf{E}}{(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} (m, l, \kappa) \xrightarrow{e} (m', l', \kappa')}$$

$$\frac{(\mathfrak{L}, \mathfrak{u}) \vdash (m, l, \kappa) \to (m', l', \kappa')}{(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} (m, l, \kappa) \to (m', l', \kappa')}$$

$$\frac{\begin{array}{c}\mathsf{Kind}(l) = \texttt{Call}(f) \quad f \notin \mathrm{dom}(\mathfrak{u}) \\ \mathfrak{e} = \texttt{Extcall}(f, m, m') \\ l' = \mathsf{Restore}(m', \mathsf{Backup}(m, l))\end{array}}{(\mathfrak{L}, \mathfrak{u}) \vdash_{comp} (m, l, \kappa) \xrightarrow{\mathfrak{e}} (m', l', \kappa)} \quad (\text{EXTCALL})$$

- *As in the procedural semantics, the set of results is* **MS** *and the final configurations with result m are the configurations (m, l, ε) where $\mathsf{Kind}(l) = \texttt{Return}$.*

The compositional big-step semantics *of $\mathfrak{u}$ is the function $[\![\mathfrak{u}]\!]_{comp} : \mathrm{dom}(\mathfrak{u}) \longrightarrow \mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}_{\mathbf{F} \setminus \mathrm{dom}(\mathfrak{u})})$ obtained from big-stepping the compositional small-step semantics.*

$$[\![\mathfrak{u}]\!]_{comp}(f)(m) = (\!|\mathsf{Comp}[\mathfrak{L}, \mathfrak{u}]|\!)(m, \mathsf{Init}(\mathfrak{u}(f), m), \varepsilon)$$

5. Linking

In this section, we are going to define a *linking* operator $\bowtie$ between two partial functions from $\mathbf{F}$ to $(\mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}))$. This linking operator will be defined directly at the level of the behaviors, independent of the underlying languages that the modules are written in.

[2] Mathematically, modules and programs in $\mathfrak{L}$ are the same. But conceptually, a program is intended to be stand-alone, and is not expected to call functions that are not defined within itself, contrary to a compilation unit, which we view as an open module.

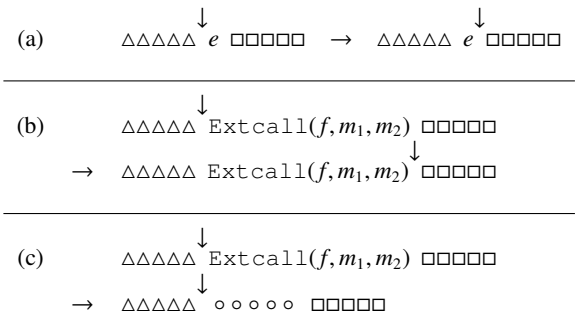


Figure 1. Three cases in behavior simulation: (a) regular event; (b) $f \notin \mathrm{dom}(\psi)$; (c) $\circ\circ\circ\circ\circ \in \psi(f)$.

Intuitively, each event corresponding to an external function call will be replaced with the behavior of the callee. However, plain straightforward substitution is not enough, as the behaviors of a compilation unit $\mathfrak{u}_1$ can involve external calls to functions defined in the other compilation unit $\mathfrak{u}_2$ that can again involve external calls to functions back in $\mathfrak{u}_1$. So, we have to *resolve* those formerly external calls that are now internal, namely the cross-calls between the two compilation units $\mathfrak{u}_1$ and $\mathfrak{u}_2$.

Let $F \subseteq \mathbf{F}$ be a set of function names. We are going to consider the functions in $\Psi(F) = F \longrightarrow (\mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}))$ that describe the behaviors of functions of F. These functions may call some "external" functions which might still be in F. We call the elements of $\Psi(F)$ *open observations*, which we usually get by taking disjoint unions of multiple compilation unit semantics.

Let ψ be such an open observation. We *resolve* the external calls (in ψ) to functions of F by recursively supplying ψ to do the substitution, yielding an observation $\mathcal{R}(\psi)$ in the set $\Phi(F) = F \longrightarrow (\mathbf{MS} \longrightarrow \mathcal{P}(\mathbb{B}_{\mathbf{F} \setminus F}))$ of *closed observations*, where there are no remaining external function call events to functions in F. We shall formally define $\mathcal{R}$ in definition 12.

Finally, if ψ_1 and ψ_2 are observations with disjoint domains, then we define the linking operator as $\psi_1 \bowtie \psi_2 = \mathcal{R}(\psi_1 \uplus \psi_2)$.

5.1 Internal call resolution by behavior simulation

Let $\psi \in \Psi(F)$ be an open observation. To resolve its internal function calls, we are going to define a semantics that will actually *simulate* the behaviors of ψ.

This resolution cancels out matching external call events by inlining each's behavior. We define this resolution by simulating the local behaviors of each module through a small-step semantics, treating each "external call" event through one "computation" step.

The simulation process is shown Fig. 1. In each case, $\downarrow$ can be seen as a cursor behind which lies the next event to be simulated. Each step $(\cdot \to \cdot)$ of the behavior simulation progresses based on the next event. All regular events are echoed as in (a), as well as all external function call events that correspond to functions not in ψ (b). By contrast, each external function call event corresponding to a function defined in ψ is replaced with the callee's events (c) where the cursor remains in the same spot ready to simulate the newly inserted events. Each step only performs one replacement at a time; the external function calls of the inlined behavior are not replaced yet until the cursor actually reaches them.

Then, we obtain the resulting linked semantics by big-stepping this small-step semantics (see examples in Fig. 2).

Consider a function f_0 and a behavior $\sigma \bullet \texttt{Extcall}(f, m_1, m_2) \cdot \mathbb{b}$ being simulated. Assume that the prefix event sequence σ has already been simulated so the `Extcall` is the first encounter of an

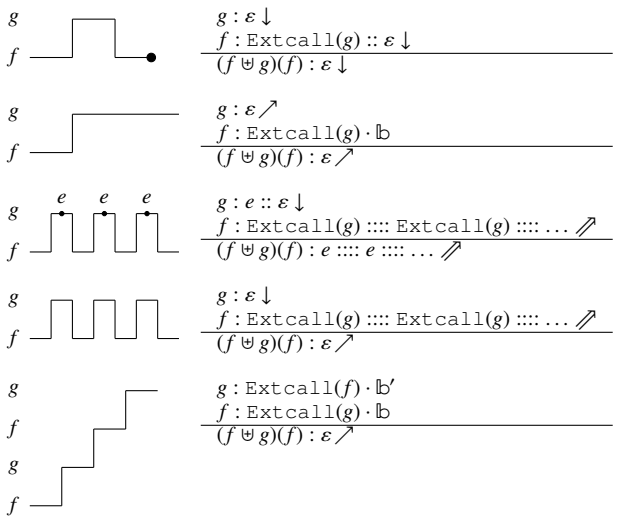


Figure 2. Examples of behaviors with external function calls between two compilation units, one defining f, another defining g, obtained by big-stepping the behavior simulation semantics. To simplify, we assume that f and g do not use any memory state.

external function call event where $f \in F$. Then m_1 is the memory state under which f is to be called, and m_2 is the expected memory state upon return of f. Now, a behavior $\mathbb{b}_2$ is chosen in $\psi(f)(m_1)$, and is to be simulated, whereas the expected return memory state m_2 as well as the remaining behavior of the caller $\mathbb{b}$ to be simulated are pushed on top of a *continuation stack*. There are three cases:

- The simulation of this chosen behavior $\mathbb{b}_2$ terminates with the expected return memory state m_2. In this case, the remaining behavior $\mathbb{b}$ of the caller f_0 after the external call, popped from the continuation stack along with m_2, can be simulated.
- The simulation of this chosen behavior $\mathbb{b}_2$ goes wrong, diverges or reacts. In such cases, the simulation result of $\mathbb{b}_2$ takes over and never returns; the remaining behavior $\mathbb{b}$ of the caller after the external call event is discarded.
- The simulation of this chosen behavior $\mathbb{b}_2$ terminates, but with a return memory state that is not m_2. In this case, the remaining behavior of the caller is discarded, too, because it was relevant only in the case of termination with m_2. Actually, it means that the simulation of the caller behavior is *spurious*. This is because the set of behaviors of the caller f_0 has a behavior $\sigma \bullet \texttt{Extcall}(f, m_1, m_2') \cdot \mathbb{b}'$ for every m_2', but most of the guesses are wrong. However, even though the simulation of the particular behavior does not make sense, the rule (EXTCALL) guarantees to have all possibilities covered, hence there will always be at least one behavior that is not spurious, it is OK to tag this irrelevant behavior as *spurious*. Formally, the simulation will not go wrong, but abruptly terminate with a special result `Spurious`. In the end, when big-stepping the small-step semantics, those spurious behaviors can be easily removed.

Definition 11 (Behavior simulation)**.** *We define the* behavior simulation small-step semantics $\mathfrak{B}[\psi]$ *as follows:*

- *The set of events is* $\mathbb{E}$.
- *The set of configurations is defined as follows:*

s	$\in$	$\mathbf{S}$		
	::=	`Spurious`		*Spurious state*
	|	$(\mathbb{b}, \chi)$	$(\mathbb{b} \in \mathbb{B},$	*Regular*
			$\chi \in (\mathbf{MS} \times \mathbb{B})^*)$	*configuration*

  That is, either a special state for spurious executions, or a normal configuration with the current behavior $\mathbb{b}$ *being simulated, paired with* χ*, the stack of the remaining expected outcomes (if the current behavior simulations terminate) and the remaining behaviors to simulate.*
- *The transition relation* $\psi \vdash \cdot \longrightarrow \cdot$ *is defined as follows:*

$$\frac{\mathbb{e} = e \in \mathbf{E}}{\psi \vdash (\mathbb{e} \cdot \mathbb{b}, \chi) \xrightarrow{\mathbb{e}} (\mathbb{b}, \chi)}$$

$$\frac{f \notin \mathrm{dom}(\psi) \quad \mathbb{e} = \texttt{Extcall}(f, m_1, m_2)}{\psi \vdash (\mathbb{e} \cdot \mathbb{b}, \chi) \xrightarrow{\mathbb{e}} (\mathbb{b}, \chi)}$$

$$\frac{\mathbb{b}' \in \psi(f)(m_1)}{\psi \vdash (\texttt{Extcall}(f, m_1, m_2) \cdot \mathbb{b}, \chi) \rightarrow (\mathbb{b}', ((m_2, \mathbb{b}) :: \chi))}$$

$$\frac{}{\psi \vdash (\varepsilon \downarrow (m), ((m, \mathbb{b}) :: \chi)) \rightarrow (\mathbb{b}, \chi)} \quad \text{(RETURN)}$$

$$\frac{}{\psi \vdash (\varepsilon \nearrow, \chi) \rightarrow (\varepsilon \nearrow, \chi)}$$

$$\frac{m' \neq m}{\psi \vdash (\varepsilon \downarrow (m'), ((m, \mathbb{b}) :: \chi)) \rightarrow \texttt{Spurious}} \quad \text{(RETURN-SPURIOUS)}$$

- *The set of results is defined as follows:*

r	$\in$	$\mathbf{R}$		
	::=	`Spurious`		*Spurious behavior*
	|	m	$(m \in \mathbf{MS})$	*Regular termination*

- *The behavior sequence* $((\varepsilon \downarrow (m)), \varepsilon)$ *is the only final state with result* $m \in \mathbf{MS}$*.* `Spurious` *is the only final state with result* `Spurious`.

Definition 12 (Resolution)**.** *Let* $\mathbb{B}^{\texttt{Spurious}} = \{\sigma \downarrow \texttt{Spurious} : \sigma \in \mathbb{E}^*\}$ *be the set of all spurious behaviors.*

Then, the resolution *of an open observation* $\psi \in \Psi(F)$ *is the closed observation* $\mathcal{R}(\psi) \in \Phi(F)$ *defined using the big-step semantics of the behavior simulation small-step semantics, excluding spurious behaviors:*

$$\mathcal{R}(\psi)(f)(m) = \bigcup_{\mathbb{b} \in \psi(f)(m)} (\!|\mathfrak{B}[\psi]|\!)(\mathbb{b}, \varepsilon) \backslash \mathbb{B}^{\texttt{Spurious}}$$

5.2 Semantic linking

Thanks to the resolution operator, we can simply define the linking of two observations:

Definition 13 (Linking)**.** *Let* ψ_1, ψ_2 *be two observations with disjoint domains. Then, their linking* $\psi_1 \bowtie \psi_2$ *is defined as:*

$$\psi_1 \bowtie \psi_2 = \mathcal{R}(\psi_1 \uplus \psi_2)$$

With the definition of linking at the level of behaviors, we can show that the compositional semantics of a compilation unit is indeed compositional. In other words, in the special case where the two modules are in the same language, linking their compositional semantics at the level of their behaviors exactly corresponds to the compositional semantics of the syntactic concatenation of the two compilation units, which conforms to the intuition of linking:

Theorem 1. *If* $\mathfrak{u}_0, \mathfrak{u}_1$ *are two compilation units with disjoint domains in the same language with function calls, then:*

$$[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{comp} = [\![\mathfrak{u}_0]\!]_{comp} \bowtie [\![\mathfrak{u}_1]\!]_{comp}$$

Proof (in Coq). • $\supseteq$: We introduce a simulation diagram: an execution step in $[\![\mathfrak{u}_0]\!]_{\mathsf{comp}} \bowtie [\![\mathfrak{u}_1]\!]_{\mathsf{comp}}$ matches at least one execution step in $[\![\mathfrak{u}_0 \uplus \mathfrak{u}_1]\!]_{\mathsf{comp}}$. In this simulation diagram, we maintain an invariant between the configuration state $(\mathbb{b}, \chi)$ in $[\![\mathfrak{u}_0]\!]_{\mathsf{comp}} \bowtie [\![\mathfrak{u}_1]\!]_{\mathsf{comp}}$ and the configuration state (m, l, κ) in

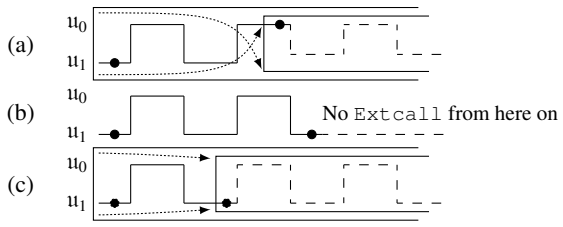


Figure 3. Illustrations of the three cases in the $\subseteq$ branch of the proof of theorem 1: (a) one of the call to u_0 never returns; (b) all external calls return with finitely many of them; (c) all external calls return with infinitely many of them. Dotted arrows denote coinduction hypothesis.

$[\![u_0 \uplus u_1]\!]_{\mathsf{comp}}$ such that κ can be decomposed in $\kappa' +\!\!+ \kappa''$ with $\mathbb{b}$ being a valid behavior in $[\![u_i]\!]_{\mathsf{comp}}$ from (m, l, κ') and κ'' matching the stack χ.

- $\subseteq$: the result for terminating and stuck behaviors is proven by induction on the length of the execution. On the other hand, diverging and reacting behaviors are dealt with in the following way. Starting from such a behavior $\mathbb{b}$ in $[\![u_0 \uplus u_1]\!]_{\mathsf{comp}}$, we first isolate an infinite step sequence corresponding to $\mathbb{b}$ by definition. Given $i \in \{0, 1\}$, we build a behavior $\mathbb{b}'$ in $[\![u_i]\!]_{\mathsf{comp}}$ by replacing the calls to functions in u_{1-i} with external calls, and we prove that simulating $\mathbb{b}'$ in $\mathfrak{B}([\![u_0]\!]_{\mathsf{comp}} \uplus [\![u_1]\!]_{\mathsf{comp}}]$ yields $\mathbb{b}$, i.e. $\mathbb{b} \in (\!|\mathfrak{B}([\![u_0]\!]_{\mathsf{comp}} \uplus [\![u_1]\!]_{\mathsf{comp}}]|\!)(\mathbb{b}', \varepsilon)$. There are three cases (each of which is illustrated in Fig. 3):
  (a) There is a call to a function in u_{1-i} that never terminates. So, before the first such external call, we can build the finite prefix of a behavior in $[\![u_i]\!]_{\mathsf{comp}}$, and deal with this external function call by coinduction replacing i with $1 - i$.
  (b) All calls to functions in u_{1-i} terminate and there are finitely many of them. So, until the last such external function call, we can build the finite prefix of a behavior in $[\![u_i]\!]_{\mathsf{comp}}$. Then, we prove that the remaining behavior of $[\![u_0 \uplus u_1]\!]_{\mathsf{comp}}$ that calls no functions of u_{1-i} is actually a behavior in $[\![u_i]\!]_{\mathsf{comp}}$.
  (c) All calls to functions in u_{1-i} terminate but there are infinitely many of them. So, we have to build a reacting behavior in $[\![u_i]\!]_{\mathsf{comp}}$ with infinitely many external function calls to u_{1-i}, each one replacing each call to a function in u_{1-i}.

□

We could have used *behavior trees* to model external function calls. Behavior trees are well-known to be used in denotational semantics to model input. They would have turned events for external function calls $\texttt{Extcall}(f, m_1, m_2)$ into branching nodes, with each branch labeled with the memory state m_2. Using behavior trees instead of plain behaviors would have helped remove spurious behaviors, as the two rules (RETURN) and (RETURN-SPURIOUS) would have been replaced by a single rule actually choosing the right branch in the behavior tree. However, this would require adopting behavior trees as the semantic object for the compositional semantics. Then, the process of making the procedural semantics into a compositional semantics would bring deep changes to the procedural small-step semantics, and the current compiler correctness proofs of CompCert based on simulation diagrams over those small-step semantics would require deep changes as well.

We believe that our current per-behavior setting, where behaviors are represented as first-order objects, shall require less intrusive changes in the current CompCert proofs. From the compiler's point of view, the external function call events introduced by our semantics need not be treated differently from ordinary events.

The relationship between the compositional semantics and the procedural semantics of a module viewed as a whole program is rather obvious: it suffices to link the compilation unit with an observation that makes every external function call stuck.

Lemma 2 (Compositional and procedural semantics)**.** *Let* $\mathfrak{u}$ *be a compilation unit in some language with function calls. Define* $\psi_{\lightning}$ *the constant stuck observation:*

$$\forall f \notin \mathrm{dom}(\mathfrak{u}), \forall m : \psi_{\lightning}(f)(m) = \{\varepsilon_{\lightning}\}$$

Then, $\forall f \in \mathrm{dom}(\mathfrak{u}) : [\![\mathfrak{u}]\!](f) = ([\![\mathfrak{u}]\!]_{comp} \bowtie \psi_{\lightning})(f)$.

6. Refinement and compiler correctness

The term "refinement" in program development dates back to the early 70s proposed by Dijkstra [5] and Wirth [21]. It quickly grows in various fields [15]. Refinement also plays a heavy role in compiler verification as shown in CompCert [12] and Müller-Olm [16].

In this work, we use refinement to define and prove correctness of separate compilation. We first state the necessary conditions for a relation to be a refinement relation. Then we show how our refinement framework applies to compiler correctness. Finally, we show that the behavior refinement relation defined in CompCert extends well to the setting of our compositional semantics.

6.1 Refinement relations

Instead of defining on pairs of programs or specifications, we define our refinement relations on sets of extended behaviors. One reason for this choice is to support refinement between multiple languages and program logics. Another reason is to better handle interactions between refinement relations and the linking operator — or, more generally, between refinement relations and the resolution operator, which we will discuss at the end of this subsection.

To generalize refinement to the compositional semantics instead of sticking to the procedural semantics of a whole program, we define refinement relations on sets of extended behaviors instead of plain behaviors.

Let $\sqsubseteq$ be a binary relation on $\mathcal{P}(\mathbb{B})$. Then, we lift it to observations in a straightforward way: we define $\psi_1 \sqsubseteq \psi_2$ if, and only if, $\mathrm{dom}(\psi_1) = \mathrm{dom}(\psi_2)$ and:

$$\forall f \in \mathrm{dom}(\psi_1), \forall m : \psi_1(f)(m) \sqsubseteq \psi_2(f)(m)$$

Definition 14 (Refinement relations)**.** *A binary relation* $\sqsubseteq$ *on* $\mathcal{P}(\mathbb{B})$ *is a refinement on observable behaviors if all these hold:*

- *reflexivity:* $\forall \mathbb{B}_0 \in \mathcal{P}(\mathbb{B}), \mathbb{B}_0 \sqsubseteq \mathbb{B}_0$
- *transitivity:* $\forall \mathbb{B}_1, \mathbb{B}_2, \mathbb{B}_3 \in \mathcal{P}(\mathbb{B})$:

$$\mathbb{B}_1 \sqsubseteq \mathbb{B}_2 \wedge \mathbb{B}_2 \sqsubseteq \mathbb{B}_3 \Longrightarrow \mathbb{B}_1 \sqsubseteq \mathbb{B}_3$$

- *congruence: for any observations* ψ_1, ψ_2 *such that* ψ_1 *is never empty (*$\forall f \in \mathrm{dom}(\psi_1), \forall m, \psi_1(f)(m) \neq \varnothing$*):*

$$\psi_1 \sqsubseteq \psi_2 \Longrightarrow \mathcal{R}(\psi_1) \sqsubseteq \mathcal{R}(\psi_2)$$

On top of a preorder, we add the congruence property, thanks to which we can easily show that refinement is compositional:

Theorem 2 (Compositionality of refinement)**.** *Let* ψ_1, ψ_2 *two observations such that* $\psi_1 \sqsubseteq \psi_2$ *and* ψ_1 *is never empty. Then, for any never-empty observation* ψ *with a domain disjoint from* ψ_1*, we have* $\psi \bowtie \psi_1 \sqsubseteq \psi \bowtie \psi_2$.

We will see in Sec. 6.4 that the CompCert improvement relation is actually a refinement relation meeting all those requirements.

6.2 Compositional program verification

Refinement is expressed at the level of extended behaviors. So, we can consider that a *specification* is an open observation, so that a compilation unit u_1 in some language with function calls can be said to make a specification ψ_1 hold if $[\![u_1]\!]_{\mathsf{comp}} \sqsubseteq \psi_1$. (The

specification ψ_1 can still contain external function call events to functions outside of $\mathrm{dom}(\mathfrak{u}_1)$.)

Thanks to refinement compositionality, and to the fact that the linking operator is defined at the semantic level of extended behaviors, this program verification scheme is compositional and suitable for open modules. Indeed, a compilation unit $\mathfrak{u}_2$ with a domain disjoint from ψ_1 can be proven to make some specification ψ hold under assumptions that it is linked with an unknown library verifying ψ_1 independently of the actual implementation of the library: we can directly link the compilation unit $\mathfrak{u}_2$ with the specification ψ_1 and prove that $\psi_1 \bowtie [\![\mathfrak{u}_2]\!]_{\mathsf{comp}} \sqsubseteq \psi$. Then, if $[\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \sqsubseteq \psi_1$, refinement compositionality gives us the refinement proof on the linked program: $[\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \bowtie [\![\mathfrak{u}_2]\!]_{\mathsf{comp}} \sqsubseteq \psi$; finally, in the special case where $\mathfrak{u}_1$ and $\mathfrak{u}_2$ are written in the same language, we have $[\![\mathfrak{u}_1 \uplus \mathfrak{u}_2]\!]_{\mathsf{comp}} \sqsubseteq \psi$.

6.3 Verified separate compilation

In this work, we follow the common notion of compiler correctness that a compiler is correct if all possible behaviors of the target program are valid behaviors of the source program. Since language specifications often leave some decisions to compilers for flexibility, a compiler is allowed to remove behaviors. In other words, compilation is a refinement step.

Our compiler correctness definition is fairly standard except for the abstract refinement relation instead of a plain subset relation. As it uses a refinement relation on extended behaviors, compiler correctness generalizes to compiling open modules by considering their compositional semantics.

Definition 15 (Compiler (optimizer) correctness)**.** *Let $\mathfrak{L}, \mathfrak{L}'$ be two languages with function calls.*

Under a refinement relation $\sqsubseteq$, a compiler C from $\mathfrak{L}$ to $\mathfrak{L}'$ is said to be correct if and only if, for any compilation unit $\mathfrak{u}$, $[\![C(\mathfrak{u})]\!]_{comp} \sqsubseteq [\![\mathfrak{u}]\!]_{comp}$.

Theorem 3. *Under a refinement relation, multiple correct compilers are compatible with separate compilation. If $\mathfrak{u}_1, \ldots, \mathfrak{u}_n$ are compilation units with disjoint domains and $C_1, \ldots, C_n$ are all correct compilers, then: $[\![C_1(\mathfrak{u}_1) \uplus \cdots \uplus C_n(\mathfrak{u}_n)]\!] \sqsubseteq [\![\mathfrak{u}_1 \uplus \cdots \uplus \mathfrak{u}_n]\!]$*

Proof. By definition of compiler correctness, $\forall i, [\![C_i(\mathfrak{u}_i)]\!]_{\mathsf{comp}} \sqsubseteq [\![\mathfrak{u}_i]\!]_{\mathsf{comp}}$. By transitivity and multiple applications of refinement compositionality (Theorem 2), we obtain

$$[\![C_1(\mathfrak{u}_1)]\!]_{\mathsf{comp}} \bowtie \ldots \bowtie [\![C_n(\mathfrak{u}_n)]\!]_{\mathsf{comp}} \sqsubseteq [\![\mathfrak{u}_1]\!]_{\mathsf{comp}} \bowtie \ldots \bowtie [\![\mathfrak{u}_n]\!]_{\mathsf{comp}}$$

which leads to $[\![C_1(\mathfrak{u}_1) \uplus \cdots \uplus C_n(\mathfrak{u}_n)]\!]_{\mathsf{comp}} \sqsubseteq [\![\mathfrak{u}_1 \uplus \cdots \uplus \mathfrak{u}_n]\!]_{\mathsf{comp}}$ because of Theorem 1 (linking in the same language). Finally, to go from the compositional to the procedural semantics, refinement compositionality with $\psi_{\mathfrak{t}}$ and Lemma 2 give the result. □

The theorem tells us that we can link several object files which are compiled independently with potentially different compilers. As long as all the compilers are correct, the linked executable will behave as an instance of the program linked at the source level.

With a single correct compiler C, Theorem 3 ensures the correctness of separate compilation even though we may not have $C(\mathfrak{u}_1) \uplus \cdots \uplus C(\mathfrak{u}_n) = C(\mathfrak{u}_1 \uplus \cdots \uplus \mathfrak{u}_n)$ (e.g. if C performs some function inlining).

6.4 Example: the CompCert refinement relation

When developing a compiler, it is usually hard or even impossible to retain one kind of behavior – the stuck behaviors. Imagine a C program that takes the address of a local variable, adds a constant to it, and then uses the result as the address to write to. If the arithmetic operation brings the address out of bound, the C semantics will get stuck. While in the target assembly code, the program is likely to continue running and crash at a much later point, or even keep going normally as the place the program writes to might be an unused stack space.

In CompCert [11], all behaviors with the event sequence before crashing as a prefix are considered "improvements" of the crashing behavior. The refinement relation it uses, initially proposed by Dockins [6] and integrated into CompCert, incorporates improvements and is an extension of a subset relation. In this section, we extend it to extended behaviors with external function call events.

Definition 16 (Behavior improvement)**.** *Let $\mathbb{b}_1, \mathbb{b}_2$ be two extended behaviors. $\mathbb{b}_1$ improves $\mathbb{b}_2$ ($\mathbb{b}_1 \sqsubseteq \mathbb{b}_2$) if and only if:*

- *either $\mathbb{b}_1 = \mathbb{b}_2$, or*
- *$\mathbb{b}_2$ is a "stuck prefix" of $\mathbb{b}_1$: there exists an event sequence σ and a behavior $\mathbb{b}$ such that $\mathbb{b}_2 = \sigma\, \mathfrak{t}$ and $\mathbb{b}_1 = \sigma \bullet \mathbb{b}$.*

Definition 17 (CompCert improvement relation)**.** *Let $\mathbb{B}_1, \mathbb{B}_2$ be two sets of extended behaviors. $\mathbb{B}_1$ improves $\mathbb{B}_2$ ($\mathbb{B}_1 \sqsubseteq \mathbb{B}_2$) if, and only if $\forall \mathbb{b}_1 \in \mathbb{B}_1, \exists \mathbb{b}_2 \in \mathbb{B}_2 : \mathbb{b}_1 \sqsubseteq \mathbb{b}_2$.*

Theorem 4. *The CompCert improvement relation is a refinement relation.*

Proof (in Coq). Congruence is proven by a lock-step backwards simulation, where the invariant between two configurations of the semantics uses behavior improvement for the behavior being simulated as well as every frame of the continuation stack. □

This theorem shows that the CompCert improvement relation defined on behaviors extends well to extended behaviors and verified separate compilation. Consequently, a correct compiler can compile an open module as if it were a whole program, by considering an external call event in no different way than a regular event. By the way, it also shows that a correct compiler necessarily preserves external function calls: in no way can it optimize them away before linking with an actual implementation for them. This is understandable because a compiler processing an open module has no hypotheses about external functions.

6.5 Coq implementation

Our Coq implementation provides the following enhancements, which we did not mention for the sake of presentation.

Functions can be passed arguments, and they can return a value. Then, the arguments are additional parameters to the semantics of a module, and they appear in the external function call events as well as the return values. Similar to the resulting memory state upon return of an external function call, the caller has to provide a behavior for each possible return value as well: given an external function f called with arguments arg and the memory state m_1, the external function call rule (EXTCALL in Def. 10) of the compositional semantics produces an event $\texttt{Extcall}(f, arg, m_1, ret, m_2)$ for any result ret and any memory state m_2.

Throughout the execution of a language with function calls, we added the ability of maintaining some *invariant* on the memory state. We equip the set of memory states with some preorder $\preceq$, such that, whenever an internal step is performed from a memory state m, the new memory state m' is such that $m \preceq m'$. Consequently, the semantics of a compilation unit provides no behavior for those external function calls that do not respect $\preceq$: in the compositional semantics, the rule for external function calls (EXTCALL in Def. 10) producing an external call event $\texttt{Extcall}(f, arg, m_1, ret, m_2)$ requires the additional premise $m_1 \preceq m_2$. This enhancement is important for CompCert, where the memory model requires that the memory evolve monotonically to prevent a deallocated memory block from being reused. The proofs of compilation passes in CompCert make critical use of this assumption.

In a language with function calls, the functions Backup and Restore which respectively save the local state into a continuation stack frame and retrieve a new one from such a frame, can change the memory state: instead of only returning a frame or local state, they return a new memory state as well. We make those functions compatible with the preorder $\preceq$ over memory states. This enhancement allows us to model the allocation and deallocation of a concrete stack frame in the memory upon function call and return.

We also provide a Coq implementation to instantiate our framework with the CompCert common subexpression elimination pass to turn it from whole-program compilation to separate compilation.

This pass is carried over CompCert RTL ("register transfer language") as both source and target languages. It is a 3-address language with infinitely many per-function-call pseudo-registers. The body of a function is a control-flow graph.

The common subexpression elimination actually replaces nodes of the control-flow graph with no-ops, if those nodes are taking part to expressions that were already computed before. This pass actually does not alter function calls and does not modify the memory between source and target programs.

We took a subset of RTL eliminating floating-point operations (due to typing constraints). Then, we added our external function call event to the CompCert so-called "external functions" (namely primitives such as volatile load and store, memory copy, or I/O, some of which generate events) to enable their support by RTL. Then, we rewrote RTL into the setting of our framework and proved that the corresponding compositional semantics and the CompCert RTL language with those new events produce the same big-step semantics. Thus, there were no changes to the proof of the compilation pass (except the removal of floating-point operations) and the correctness of separate compilation were stated directly in terms of the original RTL semantics and proved using our framework.

7. Languages with different memory state models

Our new approach is close to the way how CompCert [12] handles I/O events. Actually, we generalize it to arbitrary external function calls, and we give the formal argument why this approach is correct by enabling those external functions to be implemented and their behaviors inlined. This means that the compiler correctness techniques used for CompCert and restricted to whole programs can be easily applied to open modules.

The main difference introduced by considering the behaviors of open modules is that now part of the memory state becomes observable. There still remains a problem: a compilation pass can alter the observable memory state.

But alterations can deeply involve the structure of the memory state so that the relation between the memory states of the source programs and the compiled ones can itself change during execution. Such relations are called *Kripke worlds* [2, 10] in the setting of Kripke logical relations. But it becomes necessary to define the refinement relation as a "binary" simulation diagram deprecating the notion of the "unary" semantics.

In this section, we show that such Kripke logical relations are not necessary to deal with critical memory-changing passes of CompCert. To this purpose, we introduce a lightweight infrastructure to deal with memory-changing relations, α-refinement, that can directly cope with our unary semantics for open modules with traces of external function call events.

7.1 α-refinement

In practice, a separate compiler does make some assumptions on the behaviors of external functions. If these assumptions are also preserved as an invariant by the execution of functions defined in $\mathfrak{u}$, the compilation of $\mathfrak{u}$ can take advantage of this invariant.

Consider a module $\mathfrak{u}$ written in a procedural language $\mathfrak{L}$. Let MS be the set of memory states of $\mathfrak{L}$. Let $I \subseteq MS$ be an *invariant* in $\mathfrak{L}$, i.e. such that for any local transition $(m, l) \to (m', l')$, if $m \in I$ then $m' \in I$. Then we can restrict the set of memory states of $\mathfrak{L}$ to I, yielding a procedural language $\mathfrak{L}|_I$ such that the corresponding *compositional* semantics mandates all external function calls to return with memory states also satisfying the invariant. In other words, for any external function call event $\texttt{Extcall}(f, m_1, m_2)$ produced by the compositional semantics of $\mathfrak{L}|_I$, we always have $m_1, m_2 \in I$.

Now consider a target procedural language $\mathfrak{L}'$ having an invariant I'. Let C be a compiler from $\mathfrak{L}$ to $\mathfrak{L}'$. Then, we say that $C(\mathfrak{u})$ *α-refines* $\mathfrak{u}$ ($C(\mathfrak{u}) \sqsubseteq_\alpha \mathfrak{u}$) if, and only if there exists a bijection α between I and I' such that $[\![C(\mathfrak{u})]\!]_{\mathsf{comp}}|_{I'} \sqsubseteq \alpha([\![\mathfrak{u}]\!]_{\mathsf{comp}}|_I)$.

In practice, it means that the separate compiler C is correct when the modules are linked with other modules also satisfying the same invariants (I in the source, I' in the target). Indeed, in the case when such a bijection α exists, then we can define the procedural language $\alpha(\mathfrak{L}|_I)$ isomorphic to $\mathfrak{L}|_I$ where the set of memory states is $\alpha(I) = I'$, and then we can use the usual non-memory-changing refinement relation between $\alpha(\mathfrak{L}|_I)$ and $\mathfrak{L}'|_{I'}$. Then, separate compilation is correct provided that, when building the whole program by linking with a module containing a `main` entry point, the initial memory state passed to `main` also satisfies the invariant (I in the source, I' in the target).

Then, Theorem 3 can be rephrased as follows: if $\mathfrak{u}_1, \ldots, \mathfrak{u}_n$ are compilation units in languages $\mathfrak{L}_1, \ldots, \mathfrak{L}_n$ with disjoint domains and $C_1, \ldots, C_n$ are all compilers to the same target language $\mathfrak{L}'$ such that, for each i, C_i is correct with respect to an α_i-refinement, then:

$$[\![C_1(\mathfrak{u}_1) \uplus \cdots \uplus C_n(\mathfrak{u}_n)]\!] \sqsubseteq \alpha_1([\![\mathfrak{u}_1]\!]) \bowtie \ldots \bowtie \alpha_n([\![\mathfrak{u}_n]\!])$$

In the rest of this section, we show how to systematically turn CompCert-style memory injection into α-bijection by using a critical memory-changing pass of CompCert as an example. The same technique can also be used to support translation of calling conventions (e.g., mapping local variables or temporaries in the source into stack entries in the target).

7.2 Case study: memory injection for local variable layout

One of the most critical memory-changing compilation phases in CompCert is the phase that lays out local variables into a stack frame. Indeed, CompCert does not represent memory as a unique byte array, but as a collection of byte arrays called *memory blocks*. The purpose of this memory model is to allow pointer arithmetic only within the same block. In this setting, CompCert defines the semantics of a subset of C by allocating one block for each local variable, so that the following code example indeed gets stuck (has no valid semantics, which corresponds to *undefined behavior* according to the C standard):

```
void f (void)
{ int a[2] = {18, 42}, b[2] = {1729, 6};
  register int *pa = &a[2], *pb = &b[0];
  *pa = 3; /* undefined behavior,
              NOT equivalent to *pb = 3 */ }
```

In this example, upon function entry, CompCert allocates two different memory blocks, one (say with identifier 2) of size 8 for `a` and one (say with identifier 3) of size 4 for `b`. Then, the pointer `pa` contains an address which is, in CompCert, not a plain integer, but a *pair* $\mathsf{Vptr}(b, o)$ of the block identifier b and the byte offset o within this block. So, the value of `pa` is actually $\mathsf{Vptr}(2, 8)$ whereas `pb` is $\mathsf{Vptr}(3, 0)$. So, the two pointers are not equal, and in fact, `pa` is not a valid pointer to store to, because the size of the block identifier corresponding to `a` is 8. In other words, the boundary of one block is in no way related to other blocks. This instrumented semantics

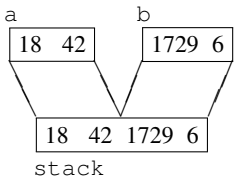


Figure 4. Injecting two arrays into the stack

can help in tracking out-of-bounds array accesses in a C program (for instance using the reference interpreter included in CompCert to "animate" the formal semantics of CompCert C).

But in practice, this C code is actually compiled by CompCert to an intermediate language called Cminor, which performs pointer arithmetic and memory operations for stack-allocated variables of a given function call in one single memory block, called the stack frame. The compiled Cminor code looks like the following C code:

```
void f (void)
{ char* stk[16];
  *(int*)(&stk[0]) = 18;   *(int*)(&stk[4]) = 42;
  *(int*)(&stk[8]) = 1729; *(int*)(&stk[12]) = 6;
  { register int* pa = (int*)(&stk[8]);
    register int* pb = (int*)(&stk[8]);
    *pa = 3;                                    }}
```

The proof of the Cminor code generation, from the Csharpminor intermediate language still having one memory block for each local variable, is based on a memory transformation called *memory injection* [13]. An injection is a partial function $\iota : \mathit{BlockID} \to (\mathit{BlockID} \times \mathbb{Z})$ mapping a source memory block to an offset within a target memory block. In our example, the target memory block allocates a stack frame (say with block-id 2) of size 16 bytes; the source memory block for `a` is mapped to offset 0 within this stack block, and `b` is mapped to offset 8: $\iota(2) = (2, 0)$ and $\iota(3) = (2, 8)$.

7.3 Issues

Although the CompCert memory injection is the most critical memory transformation used in CompCert and makes formal proofs of whole-program compilation fairly understandable (but by no means straightforward), it has several issues that make it difficult to turn those proofs into separate compilation (in the sense that it is difficult to turn the memory injection into a bijective memory transformation amenable to α-refinement).

Granularity of preservation by memory operations In the current correctness proof of the Csharpminor-to-Cminor pass, the memory injection is kept as an invariant, but the preservation properties make the memory injection hold even during the allocation of memory blocks corresponding to the source local variables. More precisely, assume that `main` is called from source memory m_0 related to target memory m'_0 by a memory injection ι_0. Then:

1. First, the stack frame block b' is created in the target memory which becomes m'_1. Memory injection ι_0 still holds between m_0 and m'_1.
2. Then, the memory block for the local variable `a`, say b_2, is created in the source memory which becomes m_2. Memory injection between m_2 and m'_1 becomes $\iota_2 = \iota_0 \uplus (b_2 \mapsto (b', 0))$.
3. Then, the memory block for the local variable `b`, say b_3 is created in the source memory which becomes m_3, injected into m'_1 through $\iota_3 = \iota_2 \uplus (b_3 \mapsto (b', 8))$

The current memory injection invariant is too fine-grained because it also holds in the middle of allocating the memory blocks for the source local variables. It actually means that the target memory m'_1 is related to any source memory that can be obtained in the middle of the allocation of such source blocks, which prevents the injectivity of the memory transformation. Conversely, the allocation of the target stack frame block is performed without changing the source memory, so that the memory injection is not even functional.

To remedy this problem, we make the preservation lemma for memory injection more coarse-grained: instead of specifying a per-allocation preservation property, we specify an all-in-one preservation property to reestablish injection only after all the blocks corresponding to source local variables are allocated.

Dynamic memory changes The proofs of compilation passes involving memory injections build the block mapping on the fly during the execution of the program: whenever a block is allocated, the mapping is modified accordingly. But the mapping is not yet known for those source memory blocks that are not allocated yet, e.g. in future function calls, or heap allocations (malloc and free library functions). It means that the mapping dynamically changes during the execution of a program. This is why Kripke logical relations are used to handle memory-changing compilation passes.

To solve those issues, we propose to define a stronger notion of memory injection in two steps. First, the block mapping is computed from the source memory using additional information contained in block tags. Then, the target memory is computed from both the source memory and the computed block mapping.

7.4 Our approach

In fact, the memory transformation for the Csharpminor-to-Cminor is actually systematic and can be defined directly depending on the shape of the memory itself rather than specified by an invariant preserved by memory operations such as allocating a new block. To this purpose, we need to add more information into the memory under the form of *tags* attached to each memory block. Such information is provided by the language semantics when allocating a new memory block, and no longer changes during the execution of the program. It plays little active role in the execution of the program, as it is only used during the compilation proof.

Block identifiers To make proofs simpler, we modify the semantics of Csharpminor and Cminor to keep the block identifiers synchronized so that as many blocks are "allocated" in the source as in the target. In the source, an empty block (within which no operation or pointer arithmetic is valid) is first allocated, then the blocks for local variables are allocated; whereas in the target, the stack frame block is first allocated with its size, then many empty blocks are allocated, one for each variable.

This has no incidence on performance: such empty blocks can be considered as *logical information*, which correspond to no memory in practice. They are not even reachable in the program.

Tags A block has a tag of one of the following forms:

$t \in T ::=$ Heap
: global variable or free store

| Stack(Main(f, sz))
: Stack frame for function f of size sz bytes

| Stack(Var(f, id, b, sz, of))
: Local variable id in f of size sz injected into b at offset of

Information defined in the tags is provided either by the semantics of Csharpminor (e.g. the identifier b' of the corresponding Main block in the tags of Var blocks) or by a previous compilation phase (e.g. offsets) within Csharpminor without changing the actual contents of memory blocks.

Specification of injection We can now replace CompCert memory injection with a stronger injection $\mathsf{INJ}(\iota, m, m')$ between a source memory m and a target memory m' axiomatized as follows:

- The empty memory injects into itself with $\iota = \varnothing$.
- If $\mathsf{INJ}(\iota, m, m')$, then $\mathsf{INJ}(\iota \uplus (b \mapsto (b, 0), m \uplus \{b\}, m' \uplus \{b\}))$ for any allocation of a new block b with tag Heap
- If $\mathsf{INJ}(\iota, m, m')$, then, if the source allocates one empty block b with tag $\mathsf{Stack}(\mathsf{Main}(f, sz))$ and several blocks corresponding to the local variables of f with tags $\mathsf{Stack}(\mathsf{Var}(f, id_i, sz_i, of_i))$ so that $[of_i, of_i + sz_i)$ are a partition of $[0, sz)$, then the target memory allocating one block b of size sz and as many empty blocks is related to the resulting source memory by INJ with $(\iota \uplus \{(b + i \mapsto (b, of_i)) : 1 \leq i \leq n\})$.
- Load, store and free operations are preserved with respect to ι
- If $\mathsf{INJ}(\iota, m, m_1)$ and $\mathsf{INJ}(\iota, m, m_2)$, then $m_1 = m_2$
- If $\mathsf{INJ}(\iota_1, m_1, m)$ and $\mathsf{INJ}(\iota_2, m_2, m)$, then $(\iota_1, m_1) = (\iota_2, m_2)$
- The block tags of the source and target memories are the same

Then, the memory transformation is defined as the partial injective function $\alpha(m) = \{m' : \exists \iota, \mathsf{INJ}(\iota, m, m')\}$. Then, we change the forward simulation proof of the CompCert Csharpminor-to-Cminor pass by replacing the injection with INJ, which incidentally proves that, actually, Csharpminor makes the invariant $\mathrm{dom}(\alpha)$ hold.

Implementation To realize those axioms, we use information contained in tags to first compute the block mapping ι from the source memory m. For any block identifier b, if the tag of b in m is $\mathsf{Stack}(\mathsf{Main}(\ldots))$, then $\iota(b)$ is undefined; if the tag of b in m is $\mathsf{Stack}(\mathsf{Var}(f, id, b', sz, of))$, then $\iota(b) = (b', of)$; otherwise, $\iota(b) = (b, 0)$.

Then, we must assume that the memory m is *well-formed*: for any block b of m of tag $\mathsf{Stack}(\mathsf{Main}(\ldots))$, this block is empty, there are no pointer to it anywhere in the memory, and it is followed by exactly the right number of blocks of tag $\mathsf{Stack}(\mathsf{Var}(f, id, b, sz, of))$ corresponding to the local variables of f and whose valid offsets are located at offsets between 0 and sz. This well-formedness condition is actually an invariant satisfied by the source language, and it will be the domain of α.

From such a memory, we can now construct the target memory m' from the memory m as follows, by scanning it from the first block. Assuming we treated all blocks between 1 and $b - 1$, we treat block b and following as follows:

- if b is the identifier of the next block available for allocation[3], then we are done.
- Otherwise, the block b is well-defined. If b is a heap block, then copy its contents (transforming pointers by ι) to the target block with the same identifier b, and move to next block $b + 1$
- Otherwise, the block b is necessarily of tag $\mathsf{Stack}(\mathsf{Main}(f, sz))$, and is empty, and its next blocks correspond to the local variables of f (say that there are n of those). Then, in the target memory m', b will have size sz and receive the contents (accordingly transforming pointers by ι) of the following blocks of m at the offsets specified by their tags; but in m', those blocks will be left empty. Then move to the next block $b + n + 1$.

Contrary to CompCert memory injections, there are no additional memory locations in the target that do not correspond to any source memory locations. This is enabled by the fact that we also add alignment constraints along with block tags to prevent alignment padding. For the sake of brevity, we do not explain this issue here.

[3] A memory always has finitely many blocks, and the number of blocks always increases because freed locations are never reused, so that a freed block is never actually deleted (only its locations are turned into unusable ones) and any newly allocated block is always fresh.

8. Related work and conclusions

Our compositional semantics is designed primarily for C-like languages, so it is not directly applicable to ML-like functional languages which have more sophisticated semantic models. C-like languages support first-class function pointers, but they do not allow function terms (e.g., $\lambda x.e$) as first-class values. C-like languages also support intensional operations such as equality test on function pointers, so it is unsound to replace one function pointer with another even if they point to functions with same observable behaviors. This allows us to use much simpler semantic objects (e.g., memory blocks with code pointers as in CompCert [13]) than sophisticated models developed for functional languages [2, 10, 1].

Compositional trace/game semantics Our idea of modeling the behavior of each external function call as an $\texttt{Extcall}(f, m, m')$ event (see Sec. 4) resembles similar treatments in compositional trace or game semantics [4, 8]. Brookes's transition-trace semantics [4] models environment transitions for shared memory concurrent languages. Under Brookes's semantics, a thread's behavior is described as a set of transition traces, with each consisting of a sequence of state transition steps $(m_1, m_1') :: (m_2, m_2') :: \ldots :: (m_n, m_n')$. The gaps between consecutive steps (e.g., m_1' and m_2, or m_{n-1}' and m_n) signal those state transitions made by other threads in the environment. Composing two threads involves calculating all the interleavings of pairs of transition traces (one from each thread) and their stuttering and mumbling closures.

Our $\texttt{Extcall}(f, m, m')$ event also uses a pair of memory states (m, m') to signal state transitions made by the environment (i.e., external calls). Our semantic linking operation (see Sec. 5) also does the "merging" of multiple event traces, but it requires more sophisticated substitutions (on behaviors) since we must also support divergence, I/O events, and reacting behaviors. It does not require stuttering and mumbling closure since we are only dealing with sequential languages. The proximity between these two approaches shows great promise toward combining these two techniques to build compositional models for concurrent C-like languages.

Ghica and Tzevelekos [8] developed a system-level semantics for composing C-like program modules. They also used external call and return events and used them to model open C-like modules and their environments. Our work can be viewed as an adaptation of their idea to the setting of compositional compiler correctness, with the goal of addressing language-independent behavior specifications that include divergence, I/O and reactive events.

Compositional CompCert Concurrently with our work, Stewart *et al* [19, 3] have recently completed the development of a formally verified separate compiler for CompCert C. This is a very impressive achievement since their Coq implementation includes all 8 translation phases from CompCert Clight to CompCert x86 plus many of the optimization phases. They developed *interaction semantics* which is a protocol-oriented operational semantics of intermodule (or thread) interaction: an open module would take normal unobservable steps or make internal function calls (defined in the same module), but would "block" when calling external functions; each such "block" point is considered as an interaction point; the program will resume execution when the external function call returns. To support both vertical and horizontal composition, they have also developed a new form of "structured simulations" which extends CompCert-style memory injections with fine-grained subjective invariants and a leakage protocol.

While our `Extcall`-event-based semantics (EES) shares many similarities to Stewart *et al*'s interaction semantics (IS), they also have some significant differences. EES does not rely on any new "protocol-oriented" operational semantics, instead, it just treats external function calls as regular events, thus it can use the same trace-based behavior specifications as semantic objects. When link-

ing two modules $\mathfrak{u}_0$ and $\mathfrak{u}_1$, our semantic linking operator $\bowtie$ (under EES) would automatically calculate the resulting semantic objects for the linked module $(\mathfrak{u}_0 \uplus \mathfrak{u}_1)$, replacing all cross-module calls between $\mathfrak{u}_0$ and $\mathfrak{u}_1$ with their corresponding behavior specifications. This leads to a very nice linking theorem (see Theorem 1 in Sec. 5): if $\mathfrak{u}_0$ and $\mathfrak{u}_1$ are two modules in the same language, linking their compositional semantics at the level of their behaviors exactly corresponds to the compositional semantics of their syntactic concatenation of the two modules. The interaction semantics (IS), on the other hand, does not attempt to "big-step" the cross-module calls between $\mathfrak{u}_0$ and $\mathfrak{u}_1$ during linking, thus it has not been able to prove the same linking theorem as we have done.

Kripke logical relations Kripke Logical Relations (KLRs) [17] are designed to support horizontal composition for functional languages. They define equivalence between terms (and values) in such a way that two functions f_1 and f_2 (of same type) are equivalent if, and only if, for any two equivalent values v_1, v_2 of the same type, $(f_1\ v_1)$ and $(f_2\ v_2)$ are equivalent. Ahmed *et al* [1, 7] showed how to generalize KLRs to reason about higher-order states. Hur and Dreyer [9] rely on step-indexed logical relations to show how to support horizontal composition; they prove correctness of a one-pass compiler but they do not support vertical composition since step-indexed logical relations are known to be not transitive.

C-like languages support both first-class function pointers and states but they do not support first-class function terms as in most functional languages. Because C function pointers can be tested for equality, a function pointer can not be replaced by another, even if they point to functions that have same observable behaviors. This is why we can build much simpler semantic models and how our new compositional semantics can still establish the monotonicity (congruence) result of our refinement relation (Section 6, Theorem 2).

Parametric bisimulations Hur et al. [10] recently proposed a promising approach that combines KLRs with bisimulations. The main idea is to abandon step-indexing but rely, instead, on coinductive simulation-based techniques (which are closer to CompCert-style simulation relations). More specifically, they propose to parameterize the local knowledge of functions with the global knowledge of external functions, and to define equivalence for open modules based on a simulation diagram over the small-step semantics of the two underlying languages of the programs. A simulation diagram can make two equivalent programs perform several steps from two equivalent states to two states corresponding to an external function call, then resume simulation upon return of such a call. This "disruption" in the flow of the simulation is analogous to our way of making the external function call explicit as a specific event in the behavior. Thus, our work can be seen as a *unary version* of their parametric bisimulations by defining a unary semantics for open modules but at the level of behaviors (independently of the small-step semantics of the underlying languages). Our way of defining the linking operator at the semantic level of behaviors avoids the need of strong typing, which makes our approach more amenable to support weakly typed C-like languages.

Conclusions In this paper, we have presented a novel compositional semantics for reasoning about open modules and for supporting verified separate compilation and linking. To build compositional semantics for open concurrent programs, we plan to split our single `Extcall` event into separate call and return events. Semantics for open concurrent programs can then have interleaving external call and return events. Semantic substitutions in our linking will be replaced by some form of "zipping" operations.

Acknowledgments We thank anonymous referees for their helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, NSF grant 1065451, and ONR Grant N00014-12-1-0478. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proc. 36th ACM Symposium on Principles of Programming Languages*, pages 340–353, 2009.

[2] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. 2009 ACM SIGPLAN International Conference on Functional Programming*, pages 97–108, 2009.

[3] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *Proc. 2014 European Symposium on Programming (ESOP'14)*, volume 8410 of *LNCS*, pages 107–127. Springer-Verlag, Apr. 2014.

[4] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996.

[5] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8:174–186, 1968.

[6] R. Dockins. *Operational Refinement for Compiler Correctness*. PhD thesis, Princeton University, 2012.

[7] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *Proc. 24th IEEE Symposium on Logic in Computer Science*, pages 71–80, 2009.

[8] D. Ghica and N. Tzevelekos. A system-level game semantics. In *Proc. 28th Conf. on the Mathematical Foundations of Programming Semantics (MFPS)*, 2012.

[9] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proc. 38th ACM Symposium on Principles of Programming Languages*, pages 133–146, 2011.

[10] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *Proc. 39th ACM Symposium on Principles of Programming Languages*, pages 59–72, 2012.

[11] X. Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2013.

[12] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[13] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformation. *Journal of Automated Reasoning*, 2008.

[14] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[15] C. C. Morgan. *Programming from specifications, 2nd Edition*. Prentice Hall International series in computer science. Prentice-Hall, 1994.

[16] M. Müller-Olm. *Modular Compiler Verification – A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer, 1997.

[17] A. Pitts and I. Stark. Operational reasoning for functions with local state. In *HOOTS'98*, pages 227–274, 1998.

[18] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig, and Y. Fu. A compositional semantics for verified separate compilation and linking. Technical Report YALEU/DCS/TR-1494, Dept. of Computer Science, Yale University, New Haven, CT, December 2014. URL: `flint.cs.yale.edu/publications/vscl.html`.

[19] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, page (to appear), 2015.

[20] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2015.

[21] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, Apr. 1971.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. 2011 ACM Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

Deep Specifications and Certified Abstraction Layers

Ronghui Gu Jérémie Koenig Tahina Ramananandro Zhong Shao
Xiongnan (Newman) Wu Shu-Chun Weng Haozhong Zhang[†] Yu Guo[†]

Yale University [†]University of Science and Technology of China

Abstract

Modern computer systems consist of a multitude of abstraction layers (e.g., OS kernels, hypervisors, device drivers, network protocols), each of which defines an interface that hides the implementation details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Despite their obvious importance, abstraction layers have mostly been treated as a system concept; they have almost never been formally specified or verified. This makes it difficult to establish strong correctness properties, and to scale program verification across multiple layers.

In this paper, we present a novel language-based account of abstraction layers and show that they correspond to a strong form of abstraction over a particularly rich class of specifications which we call *deep specifications*. Just as *data abstraction* in typed functional languages leads to the important *representation independence* property, abstraction over deep specification is characterized by an important *implementation independence* property: any two implementations of the same deep specification must have *contextually equivalent* behaviors. We present a new layer calculus showing how to formally specify, program, verify, and compose abstraction layers. We show how to instantiate the layer calculus in realistic programming languages such as C and assembly, and how to adapt the CompCert verified compiler to compile certified C layers such that they can be linked with assembly layers. Using these new languages and tools, we have successfully developed multiple certified OS kernels in the Coq proof assistant, the most realistic of which consists of 37 abstraction layers, took less than one person year to develop, and can boot a version of Linux as a guest.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, formal methods; D.3.3 [*Programming Languages*]: Languages Constructs and Features; D.3.4 [*Programming Languages*]: Processors—Compilers; D.4.5 [*Operating Systems*]: Reliability—Verification; D.4.7 [*Operating Systems*]: Organization and Design—Hierarchical design; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Abstraction Layer; Modularity; Deep Specification; Program Verification; Certified OS Kernels; Certified Compilers.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3330-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676975

1. Introduction

Modern hardware and software systems are constructed using a series of abstraction layers (e.g., circuits, microarchitecture, ISA architecture, device drivers, OS kernels, hypervisors, network protocols, web servers, and application APIs), each defining an interface that hides the implementation details of a particular set of functionality. Client programs built on top of each layer can be understood solely based on the interface, independent of the layer implementation. Two layer implementations of the same interface should behave in the same way in the context of any client code.

The power of abstraction layers lies in their use of a very rich class of specifications, which we will call *deep specifications* in this paper. A deep specification, in theory, is supposed to capture the precise functionality of the underlying implementation as well as the assumptions which the implementation might have about its client contexts. In practice, abstraction layers are almost never formally specified or verified; their interfaces are often only documented in natural languages, and thus cannot be rigorously checked or enforced. Nevertheless, even such informal instances of abstraction over deep specifications have already brought us huge benefits. Baldwin and Clark [1] attributed such use of abstraction, modularity, and layering as the key factor that drove the computer industry toward today's explosive levels of innovation and growth because *complex products can be built from smaller subsystems that can be designed independently yet function together as a whole*.

Abstraction and modularity have also been heavily studied in the programming language community [31, 30]. The focus there is on abstraction over "shallow" specifications. A module interface in existing languages cannot describe the full functionality of its underlying implementation, instead, it only describes type specifications, augmented sometimes with simple invariants. Abstraction over shallow specifications is highly desirable [24], but client programs cannot be understood from the interface alone—this makes modular verification of correctness properties impossible: verification of client programs must look beyond the interface and examine its underlying implementation, thus breaking the modularity.

Given the obvious importance, formalizing and verifying abstraction layers are highly desirable, but they pose many challenges:

- *Lack of a language-based model.* It is unclear how to model abstraction layers in a language-based setting and how they differ from regular software modules or components. Each layer seems to be defining a new "abstract machine;" it may take an existing set of mechanisms (e.g., states and functions) at the layer below and expose a different view of the same mechanisms. For example, a virtual memory management layer—built on top of a physical memory layer— would expose to clients a different view of the memory, now accessed through virtual addresses.
- *Lack of good language support.* Programming an abstraction layer formally, by its very nature, would require two languages: one for writing the layer implementation (which, given the low-

level nature of many layers, often means a language like C or assembly); another for writing the formal layer specification (which, given the need to precisely specify full functionality, often means a rich formal logic). It is unclear how to fit these two different languages into a single setting. Indeed, many existing formal specification languages [34, 18, 16] are capable of building accurate *models* with rich specifications, but they are not concerned with connecting to the actual running code.

- *Lack of compiler and linking support.* Abstraction layers are often deployed in binary or assembly. Even if we can verify a layer implementation written in C, it is unclear how to compile it into assembly and link it with other assembly layers. The CompCert verified compiler [19] can only prove the correctness of compilation for whole programs, not individual modules or layers. Linking C with assembly adds a new challenge since they may have different memory layouts and calling conventions.

In this paper, we present a formal study of abstraction layers that tackles all these challenges. We define a *certified abstraction layer* as a triple (L_1, M, L_2) plus a mechanized proof object showing that the layer implementation M, built on top of the interface L_1 (the *underlay*), indeed faithfully *implements* the desirable interface L_2 above (the *overlay*). Here, the *implements* relation is often defined as some *simulation* relation [22]. A certified layer can be viewed as a "parameterized module" (from interfaces L_1 to L_2), *a la* an SML functor [23]; but it enforces a stronger contextual correctness property: a correct layer is like a "certified compiler," capable of converting any *safe* client program running on top of L_2 into one that has the same behavior but runs on top of L_1 (e.g., by "compiling" abstract primitives in L_2 into their implementation in M).

A regular software module M (built on top of L_1) with interface L_2 may not enjoy such a property because its client may invoke another module M' which shares some states with M but imposes different state invariants from those assumed by L_2. An abstraction layer does not allow such a client, instead, such M' must be either built on top of L_2 (thus respecting the invariants in L_2), or below L_2 (in which case, L_2 itself must be changed).

Our paper makes the following new contributions:

- We present the first language-based account of certified abstraction layers and show how they correspond to a *rigorous* form of abstraction over deep specifications used widely in the system community. A certified layer interface describes not only the precise functionality of any underlying implementation but also clear assumptions about its client contexts. Abstraction over deep specifications leads to the powerful *implementation independence* property (see Sec. 2): any two implementations of the same layer interface have contextually equivalent behaviors.
- We present a new layer calculus showing how to formally specify, program, verify, and compose certified abstraction layers (see Sec. 3). Such a layer language plays a similar role as the module language in SML [23], but its interface checking is not just typechecking or signature matching; instead, it requires formal verification of the *implements* relation in a proof assistant.
- We have instantiated the layer calculus on top of two core languages (see Sec. 4 and 5): **ClightX**, a variant of the CompCert Clight language [5]; and **LAsm**, an x86 assembly language. Both ClightX and LAsm can be used to program certified abstraction layers. We use the Coq logic [35] to develop all the layer interfaces. Each ClightX or LAsm layer is parameterized over its underlay interface, implemented using CompCert's external call mechanisms. We developed new tools and tactic libraries to help automate the verification of the *implements* relation.
- We have also modified CompCert to build a new verified compiler, **CompCertX**, that can compile ClightX abstraction layers into LAsm layers (see Sec. 6). CompCertX is novel because it can prove a stronger correctness theorem for compiling individual functions in each layer—such a theorem requires reasoning about *memory injection* [21] between the memory states of the source and target languages. To support linking between ClightX and LAsm layers, we show how to design the *implements* relation so that it is stable over memory injection.
- Using these new languages and tools, we have successfully constructed several feature-rich certified OS kernels in Coq (see Sec. 7). A certified kernel (L_{x86}, K, L_{ker}) is a verified LAsm implementation K, built on top of L_{x86}, and it *implements* the set of system calls as specified in L_{ker}. The correctness of the kernel guarantees that if a user program P runs *safely* on top of L_{ker}, running the version of P linked with the kernel K on L_{x86} will produce the same behavior. All our certified kernels are built by composing a collection of smaller layers. The most realistic kernel consists of 37 layers, took less than one person year to develop, and can boot a version of Linux as a guest.

The *POPL Artifact Evaluation Committee* reviewed the full artifact of our entire effort, including ClightX and LAsm, the CompCertX compiler, and the implementation of all certified kernels with Coq proofs. The reviewers unanimously stated that our implementation *exceeded their expectations*. Additional details about our work can be found in the companion technical report [13].

2. Why abstraction layers?

In this section, we describe the main ideas behind deep specifications and show why they work more naturally with abstraction layers than with regular software modules.

2.1 Shallow vs. deep specifications

We introduce shallow and deep specifications to describe different classes of requirements on software and hardware components. Type information and program contracts are examples of "shallow" specifications. Type-based module interfaces (e.g., ML signatures) are introduced to support compositional static type checking and separate compilation: a module M can be typechecked based on its import interface L_1 (without looking at L_1's implementation), and shown to have types specified in its export interface L_2.

To support compositional verification of strong functional correctness properties on a large system, we would hope that all of its components are given "deep" specifications. A module M will be verified based on its import interface L_1 (without looking at L_1's implementation), and shown to *implement* its export interface L_2.

To achieve true modularity, we would like to reason about the behaviors of M **solely** based on its import interface L_1; and we would also like its export interface L_2 to describe the full functionality of M while omitting the implementation details.

More formally, a deep specification captures everything we want to know about any of its implementations—it must satisfy the following important "implementation independence" property:

> **Implementation independence:** *Any two implementations (e.g., M_1 and M_2) of the same deep specification (e.g., L) should have* contextually equivalent *behaviors*.

Different languages may define such contextual equivalence relation differently, but regardless, we want that, given any *whole-program* client P built on top of L, running $P \oplus M_1$ (i.e., P linked with M_1) should lead to the same observable result as running $P \oplus M_2$.

Without implementation independence, running $P \oplus M_1$ and $P \oplus M_2$ may yield different observable results, so we can prove a specific whole-program property that holds on $P \oplus M_1$ but not on $P \oplus M_2$—such whole-program property cannot be proved based on the program P and the specification L alone.

```
typedef enum {
  TD_READY, TD_RUN,
  TD_SLEEP, TD_DEAD
} td_state;

struct tcb {
  td_state tds;
  struct tcb *prev, *next;
};

struct tdq {
  struct tcb *head, *tail;
};
// ν_tcbp and ν_tdqp
struct tcb tcbp[64];
struct tdq tdqp[64];
// κ_dequeue
struct tcb *
dequeue(struct tdq *q){
  struct tcb *head,*next;
  struct tcb *pid=null;
  if(q == null)
    return pid;
  else {
    head = q -> head;
    if (head == null)
     return pid;
    else {
     pid = head;
     next = head -> next;
     if(next == null) {
      q -> head = null;
      q -> tail = null;
     } else {
      next -> prev = null;
      q -> head = next;
     }
    }
  }
  return pid;
} ...
```

```
Inductive td_state :=
| TD_READY | TD_RUN
| TD_SLEEP | TD_DEAD.


Inductive tcb :=
| TCBUndef
| TCBV (tds: td_state)
       (prev next: Z)

Inductive tdq :=
| TDQUndef
| TDQV (head tail: Z)

Record abs:={tcbp:ZMap.t tcb;
             tdqp:ZMap.t tdq}

Function σ̂_dequeue a i :=
match (a.tdqp i) with
|TDQUndef => None
|TDQV h t =>
 if zeq h 0 then
  Some (a, 0)
 else
 match a.tcbp h with
 |TCBUndef => None
 |TCBV _ _ n =>
  if zeq n 0 then
  let q':=(TDQV 0 0) in
   Some (set_tdq a i q', h)
  else
  match a.tcbp n with
  |TCBUndef => None
  |TCBV s' _ n' =>
   let q':=(TDQV n t) in
   let a':=set_tdq a i q' in
   let b:=(TCBV s' 0 n') in
    Some (set_tcb a' n b, h)
  end
 end
end ...
```

Figure 1. Concrete (in C) vs. abstract (in Coq) thread queues

```
Definition tcb := td_state.

Definition tdq := List Z.

Record abs':={tcbp:ZMap.t tcb;
              tdqp:ZMap.t tdq}
```

```
Function σ̂'_dequeue a i :=
match (a.tdqp i) with
| h :: q' =>
  Some(set_tdq a i q', h)
| nil => None
end ......
```

Figure 2. A more abstract queue (in Coq)

Hoare-style partial correctness specifications are rarely deep specifications since they fail to satisfy implementation independence. Given two implementations of a partial correctness specification for a factorial function, one can return the correct factorial number and another can just go into infinite loop. A program built on top of such specification may not be reasoned about based on the specification alone, instead, we have to peek into the actual implementation in order to prove certain properties (e.g., termination).

In the rest of this paper, following CompCert [20], we will focus on languages whose semantics are *deterministic* relative to external events (formally, these languages are defined as both *receptive* and *determinate* [33] and they support external nondeterminism such as I/O and concurrency by making events explicit in the execution traces). Likewise, we only consider interfaces whose primitives have deterministic specifications. If L is a deterministic interface, and both M_1 and M_2 implement L, then $P \oplus M_1$ and $P \oplus M_2$ should have identical behaviors since they both follow the semantics of running P over L, which is deterministic. Deterministic specifications are thus also deep specifications.

Deep specifications can, of course, also be nondeterministic. They may contain resource bounds [6], numerical uncertainties [7],

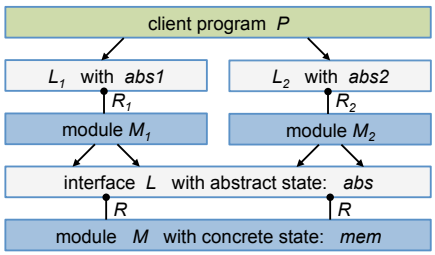


Figure 3. Client code with conflicting abstract states?

etc. Such nondeterminism should be unobservable in the semantics of a *whole* program, allowing implementation independence to still hold. We leave the investigation of nondeterministic deep specifications as future work.

2.2 Layers vs. modules

When a module (or a software component) implements an interface with a shallow specification, we often hide its private memory state completely from its client code. In doing so, we can guarantee that the client cannot possibly break any invariants imposed on the private state in the module implementation.

If a module implements an interface with a deep specification, we would still hide the private memory state from its client, but we also need to introduce an *abstract state* to specify the full functionality of each primitive in the interface.

For example, Fig. 1 shows the implementation of a concrete thread queue module (in C) and its interface with a deep specification (in Coq). The local state of the C implementation consists of 64 thread queues (tdqp) and 64 thread control blocks (tcbp). Each thread control block consists of the thread state, and a pair of pointers (prev and next) indicating which linked-list queue it belongs to. The dequeue function takes a pointer to a queue; it returns the head block if the queue is not empty, or null if the queue is empty.

In the Coq specification (Fig. 1 right; we omitted some invariants to make it more readable), we introduce an abstract state of type abs where we represent each C array as a Coq finite map (ZMap.t), and each pointer as an integer index (Z) to the tdq or tcb array. The dequeue primitive $\hat{\sigma}_{\text{dequeue}}$ is a mathematical function of type $\mathsf{abs} \to \mathsf{Z} \to \mathsf{option}\ (\mathsf{abs} \times \mathsf{Z})$; when the function returns None, it means that the abstract primitive faults. This dequeue specification is intentionally made very similar to the C function, so we can easily show that the C module indeed *implements* the specification.

We define that a module implements a specification if there is a *forward simulation* [22] from the module implementation to its specification. In the context of determinate and receptive languages [33, 20], if the specification is also deterministic, it is sufficient to find a forward simulation from the specification to its implementation (this is often easier to prove in practice).

In the rest of this paper, following CompCert, we often call the forward simulation from the implementation to its specification as *upward (forward) simulation* and the one from the specification to its implementation as *downward (forward) simulation*.

Fig. 2 shows a more abstract specification of the same queue implementation where the new abstract state abs' omits the prev and next links in tcb and treats each queue simply as a Coq list. The dequeue specification $\hat{\sigma}'_{\text{dequeue}}$ is now even simpler, which makes it easier to reason about its client, but it is now harder to prove that the C module implements this more abstract specification. This explains why we often introduce less abstract specifications (e.g., the one in Fig. 1) as intermediate steps, so a complex abstraction can be decomposed into several more tractable abstraction steps.

Deep specification brings out an interesting **new challenge** shown in Fig. 3: *what if a program P attempts to call primitives defined in two different interfaces L_1 and L_2, which may export two*

conflicting views (i.e., abstract states abs1 *and* abs2*) of the same abstract state* abs *(thus also the same concrete memory state* mem*)?*

Here we assume that modules M, M_1, M_2 implement interfaces L, L_1, L_2 via some simulation relations R, R_1, R_2 (lines marked with a dot on one end) respectively. Clearly, calling primitives in L_2 may violate the invariants imposed in L_1, and vice versa, so L_1 and L_2 are breaking each other's abstraction when we run P. In fact, even without M_2 and L_2, if we allow P to directly call primitives in L, similar violation of L_1 invariants can also occur.

This means that we must prohibit client programs such as P above, and each deep specification must state the clear assumptions about its valid client contexts. Each interface should come with a single abstract state (abs) used by its primitives; and its client can only access the same abs throughout its execution.

This is what *abstraction layers* are designed for and why they are more compositional (with respect to deep specification) than regular modules! Layers are introduced to limit interaction among different modules: only modules with identical state views (i.e., R_1, R_2 and abs1, abs2 must be identical) can be composed horizontally.

A layer interface seems to be defining a new "abstract machine" because it only supports client programs with a particular view of the memory state. The correctness of a certified layer implementation allows us to transfer formal reasoning (of client programs) on one abstract machine (the overlay) to another (the underlay).

Programming with certified abstraction layers enables a disciplined way of composing a large number of components in a complex system. Without using layers, we may have to consider arbitrary module interaction or dependencies: an invariant held in one function can be easily broken when it calls a function defined in another module. A layered approach aims to sort and isolate all components based on a carefully designed set of abstraction levels so we can reason about one small abstraction step at a time and eliminate most unwanted interaction and dependencies.

3. A calculus of abstraction layers

Motivation A user of an abstraction layer (L_1, M, L_2) wants to know that its implementation M (on top of the underlay interface L_1) can be used to run any program P written against the overlay interface L_2. If we consider L_1, L_2 as abstract machines and M as a program transformation (which transforms a program P into $M(P)$), then for some notion of refinement $\sqsubseteq$, this property can be stated as $\forall P\,.\, M(P)@L_1 \sqsubseteq P@L_2$, meaning that the behavior of $M(P)$ executing on top of the underlay specification L_1 refines that of the program P executing on top of the overlay specification L_2.

This view of abstraction layers captures a wide variety of situations. Furthermore, two layers (L_1, M, L_2) and (L_2, N, L_3) can be composed as $(L_1, M \circ N, L_3)$, and the correctness of the layer implementation $M \circ N$ follows from that of M and N.

However, the layer interfaces are often not arbitrary abstract machines, but simply instances of a base language, specialized to provide layer-specific primitives and abstract state. The implementation is not an arbitrary transformation, but instead consists of some library code to be linked with the client program. In order to prove this transformation correct, we will verify the implementation of each primitive separately, and then use these proofs in conjunction with a general template for the instrumented language.

Abstract machines and program transformations are too general to capture this redundant structure. The layer calculus presented in this section provides fine-grained notions of layer interfaces and implementations. It allows us to describe what varies from one layer to the next and to assemble such layers in a generic way.

3.1 Prerequisites

To keep the formalism general and simple, we initially take the syntax and behavior of the programs under consideration to be abstract parameters. Specifically, in the remainder of this section we will assume that the following are given:

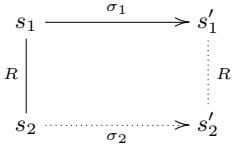
- a set of identifiers $i \in \mathbb{I}$ which will be used to name variables, functions, and primitives (e.g., dequeue and tcbp in Fig. 1);
- sets of function definitions $\kappa \in \mathrm{K}$, and variable definitions $\nu \in \mathrm{T}$, as specified by the language (e.g., $\kappa_{\mathsf{dequeue}}$ and ν_{tcbp} in Fig. 1);
- a set of behaviors $\sigma \in \Sigma$ for the individual primitives of layers, and the individual functions of programs (e.g., the step relation $\sigma_{\mathsf{dequeue}}$ derived from the Coq function $\hat{\sigma}_{\mathsf{dequeue}}$ in Fig. 1).

More examples can be found in Sec. 4.

We also need to define how the behaviors refine one another. This is particularly important because our layer interfaces bundle primitive specifications, and because a relation between layer interfaces is defined pointwise over these primitives. Ultimately, we wish to use these fine-grained layers and refinements to build complete abstract machines and whole-machine simulations. This can only be done if the refinements of individual primitives are consistent; for example, if they are given in terms of the same simulation relation.

Hence, we index behavior refinement by the elements of a partial monoid $(\mathbb{R}, \circ, \mathbf{id})$. We will refer to the elements $R \in \mathbb{R}$ of this monoid as *simulation relations*. However, note that at this stage, the elements of $\mathbb{R}$ are entirely abstract, and we require only that the composition operator $\circ$ and identity element $\mathbf{id}$ satisfy the monoid laws $R \circ (S \circ T) = (R \circ S) \circ T$ and $R \circ \mathbf{id} = \mathbf{id} \circ R = R$.

Finally, we need to interpret these abstract simulation relations as refinement relations between behaviors. That is, for each $R \in \mathbb{R}$, we require a relation $\leqslant_R$ on Σ. For instance, if the behaviors $\sigma_1, \sigma_2 \in \Sigma$ are taken to be step relations over some sets of states, $\sigma_1 \leqslant_R \sigma_2$ may be interpreted as the following simulation diagram:



That is, whenever two states s_1, s_2 are related by R in some sense, and σ_1 takes s_1 to s_1' in one step, then there exists s_2' such that σ_2 takes s_2 to s_2' in zero or more steps, and s_2' and s_1' are also related by R. The relations $\leqslant_{\_}$ should respect the monoid structure of $\mathbb{R}$, so that for any $\sigma \in \Sigma$ we have $\sigma \leqslant_{\mathbf{id}} \sigma$, and so that whenever $R, S \in \mathbb{R}$ and $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$ such that $\sigma_1 \leqslant_R \sigma_2$ and $\sigma_2 \leqslant_S \sigma_3$, it should be the case that $\sigma_1 \leqslant_{S \circ R} \sigma_3$.

3.2 Layer interfaces and modules

The syntax of the calculus is defined as follows:

$$L ::= \varnothing \mid i \mapsto \sigma \mid i \mapsto \nu \mid L_1 \oplus L_2$$
$$M ::= \varnothing \mid i \mapsto \kappa \mid i \mapsto \nu \mid M_1 \oplus M_2$$

The layer interfaces L and modules M are essentially finite maps; constructions of the form $i \mapsto \_$ are elementary single-binding objects, and $\oplus$ computes the union of two layers or modules. This is illustrated by the proof-of-concept interpretation given in the companion technical report [13]. For example, the thread queue module, shown in Fig. 1, can be defined as $M_{\mathsf{thread\_queue}} := \mathsf{tcbp} \mapsto \nu_{\mathsf{tcbp}} \oplus \mathsf{tdqp} \mapsto \nu_{\mathsf{tdqp}} \oplus \mathsf{dequeue} \mapsto \kappa_{\mathsf{dequeue}}$, while the overlay interface can be defined as $L_{\mathsf{thread\_queue}} := \mathsf{dequeue} \mapsto \sigma_{\mathsf{dequeue}}$.

The rules are presented in Fig. 4. The inclusion preorder defined on modules corresponds to the intuition that when $M \subseteq N$, any definition present in M must be present in N as well. The composition operator $\oplus$ behaves like a join operator. However, while $M \oplus N$ is an upper bound of M and N, we do not require it to be the *least* upper bound. The order on layer interfaces extends the

$\boxed{M_1 \subseteq M_2}$

$$M \subseteq M \quad \text{MLE-REFL}$$
$$\varnothing \subseteq M \quad \text{MLE-EMPTY}$$
$$M \oplus \varnothing \subseteq M \quad \text{MLE-ID-RIGHT}$$
$$(M_1 \oplus M_2) \oplus M_3 \subseteq M_1 \oplus (M_2 \oplus M_3) \quad \text{MLE-ASSOC}$$
$$M_2 \oplus M_1 \subseteq M_1 \oplus M_2 \quad \text{MLE-COMM}$$
$$M_1 \subseteq M_1 \oplus M_2 \quad \text{MLE-UB-LEFT}$$
$$M_1 \subseteq M_2 \wedge M_2 \subseteq M_3 \Rightarrow M_1 \subseteq M_3 \quad \text{MLE-TRANS}$$
$$M_1 \subseteq M_1' \wedge M_2 \subseteq M_2' \Rightarrow M_1 \oplus M_2 \subseteq M_1' \oplus M_2' \quad \text{MLE-MON}$$

$\boxed{L_1 \leqslant_R L_2}$

$$L \leqslant_{\mathbf{id}} L \quad \text{LLE-REFL}$$
$$\varnothing \leqslant_R L \quad \text{LLE-EMPTY}$$
$$L \oplus \varnothing \leqslant_{\mathbf{id}} L \quad \text{LLE-ID-RIGHT}$$
$$(L_1 \oplus L_2) \oplus L_3 \leqslant_{\mathbf{id}} L_1 \oplus (L_2 \oplus L_3) \quad \text{LLE-ASSOC}$$
$$L_2 \oplus L_1 \leqslant_{\mathbf{id}} L_1 \oplus L_2 \quad \text{LLE-COMM}$$
$$L_1 \leqslant_{\mathbf{id}} L_1 \oplus L_2 \quad \text{LLE-UB-LEFT}$$
$$L \oplus L \leqslant_{\mathbf{id}} L \quad \text{LLE-IDEMPOTENT}$$
$$L_1 \leqslant_R L_2 \wedge L_2 \leqslant_S L_3 \Rightarrow L_1 \leqslant_{S \circ R} L_3 \quad \text{LLE-TRANS}$$
$$L_1 \leqslant_R L_1' \wedge L_2 \leqslant_R L_2' \Rightarrow L_1 \oplus L_2 \leqslant_R L_1' \oplus L_2' \quad \text{LLE-MON}$$
$$\sigma_1 \leqslant_R \sigma_2 \Rightarrow i \mapsto \sigma_1 \leqslant_R i \mapsto \sigma_2 \quad \text{LLE-INTRO-PRIM}$$

$\boxed{L_1 \vdash_R M : L_2}$

$$\frac{}{L \vdash_{\mathbf{id}} \varnothing : L} \text{EMPTY}$$

$$\frac{}{L \vdash_{\mathbf{id}} i \mapsto \nu : i \mapsto \nu} \text{VAR}$$

$$\frac{L_1 \vdash_R M : L_2 \qquad L_2 \vdash_S N : L_3}{L_1 \vdash_{R \circ S} M \oplus N : L_3} \text{VCOMP}$$

$$\frac{L \vdash_R M : L_1 \qquad L \vdash_R N : L_2}{L \vdash_R M \oplus N : L_1 \oplus L_2} \text{HCOMP}$$

$$\frac{L_1 \leqslant_R L_1' \qquad L_1 \vdash_S M : L_2 \qquad L_2' \leqslant_T L_2}{L_1' \vdash_{R \circ S \circ T} M : L_2'} \text{CONSEQ}$$

Figure 4. The fine-grained layer calculus

underlying simulation preorder $\leqslant_R$ on behaviors. Compared to $\subseteq$, it should satisfy the additional property LLE-IDEMPOTENT.

The judgment $L_1 \vdash_R M : L_2$ is akin to a typing judgment for modules. It asserts that, using the simulation relation R, the module M—running on top of L_1—faithfully implements L_2. Because modules consist of code ultimately intended to be linked with a client program, the empty module $\varnothing$ acts as a unit, and can implement any layer interface L (EMPTY). Moreover, appending first N, then M to a client program is akin to appending $M \oplus N$ in one step (VCOMP). These rules correspond to the identity and composition properties already present in the framework of abstract machines and program transformations. However, the fine-grained calculus also provides a way to split refinements (HCOMP): when two different layer interfaces are implemented *in a compatible way* by two different modules on top of a common underlay interface, then the union of the two modules implements the union of the two interfaces.

This allows us to break down the problem of verifying a layer implementation in smaller pieces, but ultimately, we need to handle individual functions and primitives. The consequence rule (CONSEQ) can be used to tie our notion of behavior refinement into the calculus. However, to make the introduction of certified code possible, we need a semantics of the underlying language.

3.3 Language semantics

Assume that layers and modules are interpreted in the respective sets $\mathbb{L}$ and $\mathbb{M}$. The semantics of a module can be understood as the effect of its code has on the underlay interface, as specified by a function

$$[\![-]\!] \ : \ \mathbb{M} \to (\mathbb{L} \to \mathbb{L})$$
$$i \mapsto \nu \leqslant_{\mathbf{id}} [\![i \mapsto \nu]\!]L \quad \text{SEM-VAR}$$
$$[\![M]\!](L \oplus [\![N]\!]L) \leqslant_{\mathbf{id}} [\![M \oplus N]\!]L \quad \text{SEM-COMP}$$
$$M_1 \subseteq M_2 \wedge L_1 \leqslant_R L_2 \Rightarrow [\![M_1]\!]L_1 \leqslant_R [\![M_2]\!]L_2 \quad \text{SEM-MON}$$

Figure 5. Semantics of modules

$[\![-]\!] : \mathbb{M} \to \mathbb{L} \to \mathbb{L}$. Given such a function, we can interpret the typing judgment as:

$$L_1 \vdash_R M : L_2 \quad \Leftrightarrow \quad L_2 \leqslant_R L_1 \oplus [\![M]\!]L_1.$$

Then the properties in Fig. 5 are sufficient to ensure the soundness of the typing rules with respect to this interpretation.

Here, surprisingly, we require that the specification refine the implementation! This is because our proof technique involves turning such a *downward* simulation into the converse *upward* simulation, as detailed in Sec. 5 (Theorem 1) and Sec. 4.3. Also, we included L_1 on the right-hand side of $\leqslant_R$ to support pass-through of primitives in the underlay L_1 into the overlay L_2.

The property SEM-COMP can be understood intuitively as follows. In $[\![M]\!](L \oplus [\![N]\!]L)$, the code of M is able to use the functions defined in N in addition to the primitives of the underlay interface L, but conversely the code of N cannot access the functions of M. However, in $[\![M \oplus N]\!]L$, the functions of M and N can call each other freely, and therefore the result should be more defined. The property SEM-MON states that making the module and underlay larger should also result in a more defined semantics.

Once a language semantics is given, we introduce a language-specific rule to prove the correctness of individual functions:

$$\frac{\text{VC}(L, \kappa, \sigma)}{L \vdash_{\mathbf{id}} i \mapsto \kappa : i \mapsto \sigma} \text{FUN}$$

where the language-specific predicate $\text{VC}(L, \kappa, \sigma)$ asserts that the function body κ faithfully implements the primitive behavior σ on top of L. This rule can be combined with the rules of the calculus to build up complete certified layer implementations.

Similarly, given a concrete language semantics, we will want to tie the calculus back into the framework of abstract machines and program transformations. For a layer interface L, we will define a corresponding abstract machine meant to execute programs written in a version of the language augmented with the primitives specified in L. The program transformation associated with a module M will simply concatenate the code of M to the client program. Then, for a particular notion of refinement $\sqsubseteq$, we will want to prove that the typing judgments entail the contextual refinement property:

$$\frac{L_1 \vdash_R M : L_2}{\forall P \,.\, (P \oplus M) @ L_1 \sqsubseteq P @ L_2}$$

Informally, if M faithfully implements L_2 on top of L_1, then invocations in P of a primitive i with behavior σ in L_2, can be satisfied by calling the corresponding function κ in M.

Indeed in Sec. 4 and Sec. 5, the primitive specifications in $[\![M]\!]L$, based on step relations, are defined to reflect the possible executions of the function definitions in M. Therefore, $L_2 \leqslant_R L_1 \oplus [\![M]\!]L_1$ implies that, for any primitive implementation in M, the corresponding deep specification in L_2 refines the execution of that function definition. Hence the execution of program P with underlay L_2 refines that of $P \oplus M$ with underlay L_1 (the properties enumerated in Fig. 5 hold for a similar reason). Properties of the language (i.e., being determinate and receptive) can then be used to reverse this refinement into the desired $(P \oplus M) @ L_1 \sqsubseteq P @ L_2$.

4. Layered programming in ClightX

In this section, we provide an instantiation of our framework for a C-like language. This instantiation serves two purposes: it illustrates a common use case for our framework, showing its usability and

practicality; and it shows that our framework can add modularization and proof infrastructure to existing language subsets at minimal cost.

Our starting point: CompCert Clight Clight [5] is a subset of C and is formalized in Coq as part of the CompCert project. Its formal semantics relies on a memory model [21] that is not only realistic enough to specify C pointer operations, but also designed to simplify reasoning about non-aliasing of different variables. From the programmer's point of view, Clight avoids most pitfalls and peculiarities of C such as nondeterminism in expressions with side effects. On the other hand, Clight allows for pointer arithmetic and is a true subset of C. Such simplicity and practicality turn Clight into a solid choice for certified programming. However, Clight provides little support for abstraction, and proving properties about a Clight program requires intricate reasoning about data structures. This issue is addressed by our layer infrastructure.

4.1 Abstract state, primitives, and layer interfaces

We enable abstraction in Clight and other CompCert languages by instrumenting the memory states used by their semantics with an *abstract state* component. This abstract state can be manipulated using *primitives*, which are made available through CompCert's external function mechanism. We call the resulting language ClightX.

Abstract state and external functions The abstract state is not just a ghost state for reasoning: it does influence the outcome of executions! However, we seek to minimize its impact on the existing proof infrastructure for program and compiler verification. We do not modify the semantics of the basic operations of Clight, or the type of values it uses. Instead, the abstract state is accessed exclusively through Clight's external function mechanism.

Primitives and layer interfaces CompCert offers a notion of *external functions*, which are useful in modeling interaction with the environment, such as input/output. Indeed, CompCert models compiler correctness through traces of events which can be generated only by external functions. CompCert axiomatizes the behaviors of external functions without specifying them, and only assumes they do not behave in a manner that violates compiler correctness. We use the external function mechanism to extend Clight with our primitive operations, and supply their specifications to make the semantics of external functions more precise.

Definition 1 (Primitive specification)**.** *Let mem denote the type of memory state, and let val denote the type of concrete values. A* primitive specification *σ over the abstract state type A is a predicate on $(\mathsf{val}^* \times \mathsf{mem} \times A) \times (\mathsf{val} \times \mathsf{mem} \times A)$: when $\sigma(args, m, a, res, m', a')$ holds, we say that the primitive takes arguments $args$, memory state m and abstract state a, and returns a result res, a memory state m' and an abstract state a'.*

The type of abstract state and the set of available primitives will constitute our notion of layer interface.

Definition 2 (Layer interface)**.** *A layer interface L is a tuple $L = (A, P)$ where A is the type of abstract state, and P is the set of primitives as a finite map from identifiers to primitive specifications over the abstract state A.*

4.2 The ClightX parametric language

Syntax The syntax of ClightX (parameterized over a layer interface L) is identical to that of Clight. It features global variables (including function pointers), stack-allocated local variables, and temporary variables t . Expressions have no side effects; in particular, they cannot contain any function call. They include full-fledged pointer arithmetics (comparison, offset, C-style "arrays").

$$
\begin{array}{rcll}
e & ::= & n \mid x \mid t & \text{Constant, variable, temporary} \\
 & \mid & \&e \mid *e \mid e_1\ op\ e_2 \mid \dots &
\end{array}
$$

Statements include assignment to a memory location or a temporary, function call and return, and structured control (loops, etc.).

$$
\begin{array}{rcll}
S & ::= & e_1 = e_2 & \text{Assignment to a memory location} \\
 & \mid & t := e & \text{Assignment to a temporary variable} \\
 & \mid & t \leftarrow e(e_1, \dots) & \text{Function call} \\
 & \mid & \texttt{return}(e) & \text{Function return} \\
 & \mid & S_1; S_2 \mid \texttt{if}(e)\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}(e)\ S &
\end{array}
$$

Function calls may refer to internal functions defined as part of a module, or to primitives defined in the underlay L. However these two cases are not distinguished syntactically. In fact, the layer calculus allows for replacing primitive specifications with actual code implementation, with no changes to the caller's code.

Definition 3 (Functions, modules)**.** *A ClightX function is a tuple $\kappa = (targs, lvars, S)$, where $targs$ is the list of temporaries to receive the arguments, $lvars$ is the list of local stack-allocated variables with their sizes, and S is a statement, the function body. A* module *M is a finite map from identifiers to ClightX functions.*

Semantics Compared with Clight, the semantics of ClightX(L) adds a notion of abstract state, and permits calls to the primitives of L. We will write $L(i)(args, m, a, res, m', a')$ to denote the semantics of the primitive associated with identifier i in L.

We present the semantics of ClightX under the form of a big-step semantics. We fix an injective mapping Γ from global variables to memory block identifiers. We write $[\![e]\!](l, \tau, m)$ for the evaluation of expression e under local variables l, temporaries τ and memory state m. We write $\Gamma, L, M, l \vdash S : (\tau, m, a) \downarrow (res; \tau', m', a')$ for the semantics of statements: from the local environment l, the temporary environment τ, the memory state m, and the abstract state a, execution of S terminates and yields result res (or $\cdot$ if no result), temporary environment τ', memory state m', and abstract state a'. For instance, the rule for `return` statements is:

$$\frac{[\![e]\!](l, \tau, m) = res}{\Gamma, L, M, l \vdash \texttt{return}(e) : (\tau, m, a) \downarrow (res; \tau, m, a)}$$

We write $\Gamma, L, M \vdash f : (args; m, a) \Downarrow (res; m', a')$ to say that a function f defined either as an internal function in the module M, or as a primitive in the layer interface L, called with list of arguments $args$, from memory state m and abstract state a, returns result res, memory m' and abstract state a'.

For internal function calls, we first initialize the temporary environment with the arguments, and allocate the local variables of the callee ($\mathsf{next}(m)$ denotes the next available block identifier in memory m, not yet allocated). Then, we execute the body. Finally, we deallocate the stack-allocated variables of the callee.

$$\frac{\begin{array}{c} M(f) = ((t_1, \dots, t_n), ((x_1, sz_1), \dots, (x_k, sz_k)), S) \\ m_1 = \mathsf{alloc}(sz_k) \circ \dots \circ \mathsf{alloc}(sz_1)(m) \\ l = \varnothing[x_1 \leftarrow \mathsf{next}(m)] \dots [x_k \leftarrow \mathsf{next}(m) + k - 1] \\ \tau = \varnothing[t_1 \leftarrow v_1] \dots [t_n \leftarrow v_n] \\ \Gamma, L, M, l \vdash S : (\tau, m_1, a) \downarrow (res; \tau', m_2, a') \\ m' = \mathsf{free}(\mathsf{next}(m), sz_1) \circ \dots \circ \mathsf{free}(\mathsf{next}(m) + k - 1, sz_k)(m_2) \end{array}}{\Gamma, L, M \vdash f : (v_1, \dots, v_n; m, a) \Downarrow (res; m', a')}$$

For primitive calls, we simply query the layer interface L:

$$\frac{L(f)(args, m, a, res, m', a')}{\Gamma, L, M \vdash f : (args; m, a) \Downarrow (res; m', a')}$$

Using the function judgment, we can state the rule for function call statements as:

$$\frac{\begin{array}{c} \forall i, [\![e_i]\!](l, \tau, m) = v_i \qquad [\![e]\!](l, \tau, m) = (b, 0) \\ \Gamma(f) = b \qquad \Gamma, L, M \vdash f : (v_1, \dots, v_n; m, a) \Downarrow (res; m', a') \\ \tau' = \tau[t \leftarrow res] \end{array}}{\Gamma, L, M, l \vdash t \leftarrow e(e_1, \dots, e_n) : (\tau, m, a) \downarrow (\cdot; \tau', m', a')}$$

$$\frac{\forall i\,.\,L_1 \vdash_{\mathbf{id}} i \mapsto \kappa_i : i \mapsto \sigma'_i \qquad \dfrac{\forall i\,.\,\sigma_i \leqslant_R \sigma'_i}{\forall i\,.\,i \mapsto \sigma_i \leqslant_R i \mapsto \sigma'_i}}{\dfrac{L_1 \vdash_{\mathbf{id}} M : L'_1 \qquad L_2 \leqslant_R L'_1}{L_1 \vdash_R M : L_2}}$$

where L_1 is the underlay, the module $M = \bigoplus_i i \mapsto \kappa_i$, the intermediate layer $L'_1 = \bigoplus_i i \mapsto \sigma'_i$, and the overlay $L_2 = \bigoplus_i i \mapsto \sigma_i$.

Figure 6. Building a certified ClightX layer

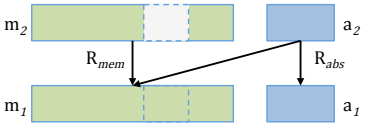


Figure 7. Layer simulation relation

The full semantics of ClightX is given in the companion TR [13].

Definition 4 (Semantics of a module)**.** *Let M be a ClightX module, and L be a layer interface. Let Γ be a mapping from global variables to memory blocks. The semantics of a module M in ClightX(L), written $[\![M]\!]L$, is the layer interface defined as follows:*

- *the type of abstract state is the same as in L;*
- *the semantics of primitives are defined by the following rule:*

$$\frac{f \in \mathrm{dom}(M) \qquad \Gamma, L, M \vdash f : (args; m, a) \Downarrow (res; m', a')}{([\![M]\!]L)(f)(args, m, a, res, m', a')}$$

4.3 Layered programming and verification

To construct a certified abstraction layer (L_1, M, L_2), we need to find a simulation R such that $L_1 \vdash_R M : L_2$ holds. Fig. 6 gives an overview of this process. We write $M = \bigoplus_i i \mapsto \kappa_i$, where i ranges over the function identifiers defined in module M, and κ_i is the corresponding implementation. Global variables in M should not be accessible from the layers above: their permissions are removed in the overlay interface L_2. The interface L_2 also includes a specification σ_i for each function i defined in M.

We decouple the task of code verification from that of data structure abstraction. We introduce an intermediate layer interface, $L'_1 = \bigoplus_i i \mapsto \sigma'_i$, with its specifications σ'_i expressed in terms of the underlay states. We first prove that $L_1 \vdash_{\mathbf{id}} M : L'_1$ holds. For each function i in M, we show that its implementation κ_i is a downward simulation of its "underlay" specification σ'_i, that is, $L_1 \vdash_{\mathbf{id}} i \mapsto \kappa_i : i \mapsto \sigma'_i$. We apply the HCOMP rule to compose all the per-function simulation statements. Note the simulation relations here are all **id**, meaning there is no abstraction of data structures in these steps. We then prove $L_2 \leqslant_R L'_1$, which means that each specification σ_i in L_2 is an abstraction of the intermediate specification σ'_i via a simulation relation R. From $i \mapsto \sigma_i \leqslant_R i \mapsto \sigma'_i$, we apply the monotonicity rule LLE-MON to get $L_2 \leqslant_R L'_1$. Finally, we apply the CONSEQ rule to deduce $L_1 \vdash_R M : L_2$.

Verifying ClightX functions L_1 and L'_1 share the same views of both concrete and abstract states, so no simulation relation is involved during this step of verification (the FUN rule in Sec. 3.3). Using Coq's tactical language, we have developed a proof automation engine that can handle most of the functional correctness proofs of ClightX programs. It contains two main parts: a ClightX statement/expression interpreter that generates the verification conditions by utilizing rules of ClightX big-step semantics, and an automated theorem prover that discharges the generated verification conditions on the fly. The automated theorem prover is a first order prover, extended with different theory solvers, such as the theory of integer arithmetic and the theory of CompCert style partial maps. The entire automation engine is developed in Coq's Ltac language.

```
typedef enum {
  PG_RESERVED, PG_KERNEL,
  PG_NORMAL
} pg_type;

struct page_info {
  pg_type t;
  uint u;
};
struct page_info AT[1<<20];
```

```
Notation RESV := 0.
Notation KERN := (RESV + 1).
Notation NORM := (KERN + 1).

Inductive page_info :=
| ATV (t: Z) (u: Z)
| ATUndef.

Record abs'' :=
  {AT: ZMap.t page_info}.
```

Figure 8. Concrete (C) vs. abstract (Coq) memory allocation table

```
// κ_at_get
uint at_get (uint i){
  uint allocated;
  allocated = AT[i].u;
  if (allocated != 0)
      allocated = 1;
  return allocated;
}

// κ_at_set
void at_set (uint i, uint b){
    AT[i].u = b;
}
```

```
Function σ̂_at_get a i :=
match (a.AT i) with
| ATV _ 0 => Some 0
| ATV _ _ => Some 1
| _ => None
end.

Function σ̂_at_set a i b :=
match (a.AT i) with
| ATV t _ =>
Some (set_AT a i (ATV t b))
| _ => None
end.
```

Figure 9. Concrete vs. abstract getter-setter functions for AT

```
Inductive σ'_at_set :=
| ∀ m m' a ofs v n,
   m.store AT ofs v = m'
  -> ofs = n * 8 + 4
  -> 0 <= n < 1048576
  -> σ'_at_set (n::v::nil)
       m a Vundef m' a.
```

```
Inductive σ_at_set :=
| ∀ m a a' n v,
   σ̂_at_set a n v = Some a'
  -> 0 <= n < 1048576
  -> σ_at_set (n::v::nil)
       m a Vundef m a'.
```

Figure 10. High level and low level specification for at_set

Data abstraction Since primitives in L'_1 and L_2 are atomic, we prove the single-step downward simulation between L'_1 and L_2 only at the specification level. The simulation proof for the abstraction can be made language independent. The simulation relation R captures the relation between the underlay state (concrete memory and abstract state) and the overlay state, and can be decomposed as R_{mem} and R_{abs} (see Fig. 7). The relation R_{mem} ensures that the concrete memory states m_1 and m_2 contain the same values, while making sure the memory permissions for the part to be abstracted are erased in the overlay memory m_2. The component R_{abs} relates the overlay abstract state a_2 with the full underlay state (m_1, a_1).

Through this decomposition, we achieve the following two objectives: the client program can directly manipulate the abstract state without worrying about its underlying concrete implementation (which is hidden via R_{mem}), and the abstract state in the overlay is actually implementable by the concrete memory and abstract state in the underlay (via R_{abs}).

Common patterns We have developed two common design patterns to further ease the task of verification. The *getter-setter* pattern establishes memory abstraction by introducing new abstract states and erasing the corresponding memory permissions for the overlay. The overlay only adds the get and set primitives which are implemented using simple memory load/store operations at the underlay. The *abs-fun* pattern implements key functionalities, but does not introduce new abstract state. Its implementation (on underlay) does not touch concrete memory state. Instead, it only accesses the states

```
// κ_palloc
uint palloc(uint nps){
  uint i = 0, u;
  uint freei = nps;
  while(freei == nps
        && i < nps) {
    u = at_get(i);
    if (u == 0)
      freei = i;
    i ++;
  }
  if (freei != nps)
    at_set(freei, 1);
  return freei;
}
```

```
Definition first_free a n:
  {v| 0<= fst v < n
   /\ a.AT (fst v) = ATV (snd v) 0
   /\ ∀ x, 0 <= x < fst v
       -> ~ a.AT x = ATV _ 0}
+ {∀ x, 0 <= x < n
       -> ~ a.AT x = ATV _ 0}.

Function σ̂_palloc a nps :=
match first_free a nps with
 | inleft (exist (i, t) _) =>
   (set_AT a i (ATV t 1), i)
 | _ => (a, nps)
end.
```

Figure 11. Concrete (in C) vs. abstract (in Coq) palloc function

```
Inductive σ'_palloc : spec :=
| ∀ m a a' nps n,
   σ̂_palloc a nps = (a', n)
  -> 0 <= nps < 1048576
  -> σ'_palloc (nps::nil) m a n m a'.

Definition σ_palloc := σ'_palloc.
```

Figure 12. High level and low level specification for palloc function

that have already been abstracted, and it only does so using the primitives provided by the underlay interface.

Figs. 8-12 show how we use the two patterns to implement and verify a simplified physical memory allocator palloc, which allocates and returns the first free entry in the physical memory allocation table. Fig. 8-10 shows how we follow the *getter-setter* pattern to abstract the allocation table into a new abstract state. As shown in Fig. 8, we first turn the concrete C memory allocation table implementation into an abstract Coq data type. Then we implement the getter and setter functions for the memory allocation table, both in C and Coq (see Fig. 9). The Coq functions $\hat{\sigma}_{\text{at\_get}}$ and $\hat{\sigma}_{\text{at\_set}}$ are just intermediate specifications that are used later in the overlay specifications. The actual underlay and overlay specifications of the setter function at_set are shown in Fig. 10.

We then prove $L_1 \vdash_{\textbf{id}} \text{at\_set} \mapsto \kappa_{\text{at\_set}} : \text{at\_set} \mapsto \sigma'_{\text{at\_set}}$, and also $\text{at\_set} \mapsto \sigma_{\text{at\_set}} \leqslant_R \text{at\_set} \mapsto \sigma'_{\text{at\_set}}$.

The code verification (first part) is easy for this pattern because the memory load and store operations in the underlay match the source code closely. The proof can be discharged by our automation tactic. The main task of this pattern is to prove refinement (second part): we design a simulation relation R relating the memory storing the global variable at underlay with its corresponding abstract data at overlay. The component R_{mem} ensures that there is no permission for allocation table `AT` in overlay memory state m_2, while the component R_{abs} is defined as follows:

- $\forall i \in [0, 2^{20})$, R_{abs} enforces the *writable* permission on `AT[i]` at underlay memory state m_1, and requires (a_2.`AT i`) at overlay to be `(ATV AT[i].t AT[i].u)`.
- Except for `AT`, R_{abs} requires all other abstract data in underlay and overlay to be the same.

The refinement proof for $L_2 \leqslant_R L_1'$ involves the efforts to prove that this relation R between underlay memory and overlay abstract state is preserved by all the atomic primitives in both L_1' and L_2.

After we abstract the memory and get/set operations, we implement palloc on top of L_2, following the *abs-fun* pattern. The previous overlay now becomes the new underlay ("L_1"). Fig. 11 shows both the implementation of palloc in ClightX and the abstract function in Coq. As before, we separately show that $L_1 \vdash_{\textbf{id}} \text{palloc} \mapsto \kappa_{\text{palloc}} : \text{palloc} \mapsto \sigma'_{\text{palloc}}$, and $\text{palloc} \mapsto \sigma_{\text{palloc}} \leqslant_R \text{palloc} \mapsto \sigma'_{\text{palloc}}$ holds. For the *abs-fun* pattern, the refinement proof is easy. Since we do not introduce any new abstract states in this pattern, the implementation only manipulates the abstract states through the primitive calls of the underlay. Thus, as shown in Fig. 12, the corresponding underlay and overlay specifications are exactly the same, so the relation R here is the identity (**id**) and the proof of refinement is trivial. The main task for the *abs-fun* pattern is to verify the code, which is done using our automation tactic.

The above examples show that for the *getter-setter* pattern, the primary task is to prove data abstraction, while for the *abs-fun* pattern, the main task is to do simple program verification. These two tasks are well understood and manageable, so the decoupling (via these two patterns) makes the layer construction much easier.

5. Layered programming in LAsm

In this section, we describe LAsm, the *Layered Assembly language*, and the extended machine model which LAsm is based on.

The reason we are interested in assembly code and behavior is threefold. First of all, even though we provide ClightX to write most code, we are still interested in the actual assembly code running on the actual machine. In Section 6, we will provide a verified compiler to transport all proofs of code written in ClightX to assembly.

Secondly, there are parts of software that have to be manually written in assembly for various reasons. For example, the standard implementation of kernel context switch modifies the stack pointer register ESP, which does not satisfy the C calling convention and has to be verified in assembly. A linker will be defined in Section 6 to link them with compiled C code.

Last but not least, we are interested not only in the behavior of our code, but also in the behavior of the *context* that will call functions defined in our code. To be as general as possible, we allow the context to include all valid assembly code sequences. To this end, it is necessary to transport per-function refinement proofs to a whole-machine *contextual refinement* proof.

The LAsm assembly language We start from the 32-bit x86 assembly subset specified in CompCert. CompCert x86 assembly is modeled as a state machine with a register set and a memory state. The register set consists of eight 32-bit general-purpose registers and eight XMM registers designated as scalar double-precision floating-point operands. The memory state is same as the one in Clight. In particular, each function executes with its stack frame modeled in its own memory block, so that the stack is not a contiguous piece of memory. Another anomaly regarding function calls in CompCert x86 assembly is that the return address is stored in pseudo-register RA instead of being pushed onto the stack, so that the callee must allocate its own stack frame and store the return address.

Similarly to ClightX, we extend the machine state with an abstract state, which will be modified by primitives. This yields LAsm, whose syntax is the same as that of CompCert x86 assembly, except that the semantics will be parameterized over the type of abstract states and the specifications of primitives. Most notably, primitive calls are syntactically indistinguishable from normal function calls, yet depend on the specifications semantically.

Moreover, in our Coq formalization, the semantics of LAsm is also equipped with memory accessors for address translation in order to handle both kernel memory linear mapping and user space virtual memory. However, for the sake of presentation, we are going to describe a simplified version of LAsm where memory accesses only use the kernel memory.

We define the semantics of LAsm in small-step form. The machine state is (ρ, m, a) where ρ contains the values of registers, m is the concrete memory state and a is the abstract state. Let M be an LAsm module, which is a finite map from identifiers to arrays of LAsm instructions, we write $\Gamma, L, M \vdash (\rho, m, a) \rightarrow (\rho', m', a')$

a transition step in the LAsm machine. The full syntax and formal semantics of LAsm is described in the companion technical report.

Assembly layer interfaces The semantics of LAsm is parameterized over a layer interface. Different from C-style primitives (see Def. 1), which are defined using argument list and return value, primitives implemented in LAsm often utilize their full control over the register set and are not restricted to a particular calling convention (e.g. context switch). Therefore, it is necessary to extend the structure of layer interfaces to allow assembly-style primitives modifying the register set.

Definition 5 (Assembly-style primitive)**.** *An assembly-style primitive specification p over the abstract state type A is a predicate on $((\mathsf{preg} \to \mathsf{val}) \times \mathsf{mem} \times A) \times ((\mathsf{preg} \to \mathsf{val}) \times \mathsf{mem} \times A)$. $p(\rho, m, a, \rho', m', a')$ says that the primitive p takes register set ρ, memory state m and abstract state a as arguments, and returns register set ρ', memory state m' and abstract state a' as result.*

By "*style*," we mean the calling convention, not the language in which they are actually implemented. C-style primitives may very well be implemented as hand-written assembly code at underlay.

We can then define assembly layer interfaces by replacing the primitive specification with our assembly-style one in Def. 2. But, to make reasoning simpler, when defining assembly layer interfaces, we distinguish C-style from assembly-style primitives. First, C-style primitives can be refined by other C-style primitives. Second and most importantly, it becomes possible to instantiate the semantics of ClightX with an *assembly* layer interface by just considering C-style primitives and ignoring assembly-style primitives (which might not follow the C calling convention). In this way, ClightX code is only allowed to call C-style primitives, whereas LAsm can actually call both kinds of primitives.

Definition 6 (Assembly layer interface)**.** *An assembly layer interface L is a tuple $L = (A, P_{\mathrm{ClightX}}, P_{\mathrm{LAsm}})$ where:*

- *$(A, P_{\mathrm{ClightX}})$ is a C layer interface (see Def. 2)*
- *P_{LAsm} is a finite map from identifiers to assembly-style primitive specifications over the abstract state A. The domains of P_{ClightX} and P_{LAsm} shall be disjoint.*

Whole-machine semantics and contextual refinement Based on the relational transition system which we just defined for LAsm, we can define the whole-machine semantics including not only the code that we wrote by hand or that we compile, but also the *context* code that shall call our functions. To this end, it suffices to equip the semantics with a notion of initial and final state, in a way similar to the CompCert x86 whole-program assembly semantics.

In CompCert, the initial state consists of an empty register set with only EIP (instruction pointer register) pointing to the `main` function of the module, and the memory state is constructed by allocating a memory block for each global variable of the program. We follow the same approach for LAsm, except that we also need an initial abstract state, provided by the layer interface, so we need to extend its definition:

Definition 7 (Whole-machine layer interface)**.** *A whole-machine layer interface L is a tuple $L = (A, P_{\mathrm{ClightX}}, P_{\mathrm{LAsm}}, a_0)$ where:*

- *$(A, P_{\mathrm{ClightX}}, P_{\mathrm{LAsm}})$ is an assembly layer interface*
- *$a_0 : A$ is the initial abstract state.*

Definition 8 (Whole-machine initial state)**.** *The whole-machine LAsm initial state for layer interface L and module M is the LAsm state (ρ_0, m_0, a_0) defined as follows:*

- $\rho_0(r) = \begin{cases} (\Gamma(\mathtt{main}), 0) & \text{if } r = \mathsf{EIP} \\ 0 & \text{if } r = \mathsf{RA} \\ \bot & \text{otherwise} \end{cases}$
- *m_0 is constructed from the global variables of Γ, L, M*
- *a_0 is the whole-machine initial state specified in L*

Definition 9 (Whole-machine final state)**.** *A whole-machine LAsm state (ρ, m, a) is final with return code n if, and only if, $\rho(\mathsf{EAX}) = n$ and $\rho(\mathsf{EIP}) = 0$, where EAX is the accumulator register.*

Notice that $\rho(\mathsf{EIP})$ contains the *integer* 0, which is also the initial return address and is not a valid pointer. This ensures that executions do not go beyond a final state, following the CompCert x86 whole-program semantics: `main` has returned to its "caller", which does not exist. Thus, the final state is uniquely determined (there can be no other possible behavior once such a state is reached), so the whole-machine semantics is deterministic once the primitives are.

Definition 10 (Whole-machine behavior)**.** *Let Γ be a mapping of global variables to memory blocks. Then, we say that*

- *$LAsm(\Gamma, L, M)$* diverges *if there is an infinite execution sequence from the whole-machine initial state for L*
- *$LAsm(\Gamma, L, M)$* terminates with return code n *if there is a finite execution sequence from the whole-machine initial state for L to a whole-machine final state with return code n*
- *$LAsm(\Gamma, L, M)$* goes wrong *if there is a finite execution sequence from the whole-machine initial state for L to a non-final state that can take no step.*

Then, we are interested in *refinement* between whole machines:

Definition 11 (Whole-machine refinement)**.** *Let L_{high}, L_{low} be two whole-machine assembly layer interfaces, and M_{high}, M_{low} be two LAsm modules. Then, we say that $M_{low}@L_{low}$ refines $M_{high}@L_{high}$, and write $M_{low}@L_{low} \sqsubseteq M_{high}@L_{high}$ if, and only if, for any Γ such that $\mathrm{dom}(L_{high}) \cup \mathrm{dom}(M_{high}) \cup \mathrm{dom}(L_{low}) \cup \mathrm{dom}(M_{low}) \subseteq \mathrm{dom}(\Gamma)$ and $LAsm(\Gamma, L_{high}, M_{high})$ does not go wrong, then (1) $LAsm(\Gamma, L_{low}, M_{low})$ does not go wrong; (2) if $LAsm(\Gamma, L_{low}, M_{low})$ terminates with return code n, then so does $LAsm(\Gamma, L_{high}, M_{high})$; (3) if $LAsm(\Gamma, L_{low}, M_{low})$ diverges, so does $LAsm(\Gamma, L_{high}, M_{high})$.*

In our Coq implementation, we actually formalized the semantics of LAsm with a richer notion of observable behaviors involving CompCert-style events such as I/O. Thus, we define the whole-machine behaviors and refinement using event traces *a la* CompCert [20, 3.5 sqq.]: if the higher machine does not go wrong, then every valid behavior of the lower machine is a valid behavior of the higher.

Finally, we can define *contextual refinement* between layer interfaces through a module M:

Definition 12 (Contextual refinement)**.** *We say a module M implements an overlay L_{high} on top of an underlay L_{low}, and write $L_{low} \vDash M : L_{high}$ if, and only if, for any module (*context*) M' disjoint from M, L_{low}, L_{high}, we have $(M \oplus M')@L_{low} \sqsubseteq M'@L_{high}$.*

Per-module semantics As for ClightX, we can also specify the semantics of an LAsm module as a layer interface. However, a major difference between ClightX and LAsm is that it is not possible to uniquely characterize the "per-function final state" at which function execution should stop. Indeed, as in LAsm there is no control stack, when considering the per-function semantics of a function f, it is not possible to distinguish f exiting and returning control to its caller, from a callee g returning to f.

Thus, even though both the step relation of the LAsm semantics and the primitive specifications (of a layer interface) are deterministic, the semantics of a function could still be non-deterministic.

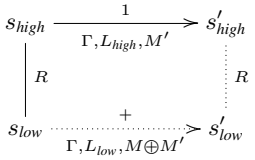
Definition 13. *Let $L = (A, \_, \_)$ be an assembly layer interface, and M be an LAsm module. The module semantics $[\![M]\!]L$ is then the assembly layer interface $[\![M]\!]L = (A, \varnothing, P)$, where the assembly-style primitive specification P is defined for each $f \in \mathrm{dom}(M)$*

using the small-step semantics of LAsm as follows:

$$\begin{array}{ll} & P(f)(\rho, m, a, \rho', m', a') \\ \Leftrightarrow & \Gamma(f) = b \wedge \rho(\mathsf{EIP}) = (b, 0) \\ & \wedge \Gamma, L, M \vdash (\rho, m, a) \rightarrow^{+} (\rho', m', a') \end{array}$$

Soundness of per-module refinement In this paper, we aim at showing that the layer calculus given in Section 3 is a powerful device to prove contextual refinement: instead of proving the whole-machine contextual refinement directly, we only need to prove the *downward simulation* relations about individual modules, notated as $L_{\text{low}} \vdash_R M : L_{\text{high}}$, and apply the soundness theorem to get the contextual refinement properties at the whole-machine level.

Lemma 1 (Downward simulation diagram)**.** *Let (L_{low}, M, L_{high}) be a certified layer, such that $L_{low} \vdash_R M : L_{high}$. Then, for any module M', we have the following downward simulation diagram:*



Theorem 1 (Soundness)**.** *Let (L_{low}, M, L_{high}) be a certified layer. If the primitive specifications of L_{low} are deterministic and if $L_{low} \vdash_R M : L_{high}$, then $L_{low} \vDash M : L_{high}$.*

Proof. Since the whole machine $LAsm(\Gamma, L_{\text{low}}, M)$ is deterministic, we can flip the downward simulation given by Lemma 1 to an upward one, hence the whole-machine refinement. □

Since the per-function semantics is non-deterministic due to its final state not being uniquely defined, we can only flip the downward simulation to contextual refinement at the whole-machine level.

6. Certified compilation and linking

We would like to write most parts of our kernel in ClightX rather than in LAsm for easier verification. This means that, for each layer interface L, we have to compile our ClightX(L) source code to the corresponding LAsm(L) assembly language in such a way that all proofs at the ClightX level are preserved at the LAsm level.

This section describes how we have modified the CompCert compiler to compile certified C layers into certified assembly layers. It also talks about how we link compiled certified C layers with other certified assembly layers.

6.1 The CompCertX verified compiler

To transport the proofs at ClightX down to LAsm, we adapt the CompCert verified compiler to parameterize all its intermediate languages over the layer interface L similarly to how we defined ClightX(L), including the assembly language. This gives rise to CompCertX(L) (for "CompCert eXtended", where external functions are instantiated with layer interface L).

CompCertX goes from ClightX to the similarly parameterized AsmX and then to LAsm. We retain all features and optimizations of CompCert, including function inlining, dead code elimination, common subexpression elimination, and tail call recognition.

Compiler correctness for CompCertX Because CompCert only proves semantics preservation for whole programs, the major challenge is to adapt the semantics preservation statements of all compilation passes (from Clight to assembly) to per-function semantics.

The operational semantics of all CompCert languages are given through small-step transition relations equipped with sets of whole-program initial and final states, so we have to redesign those states to per-function setting. For the initial state, whereas CompCert constructs an initial memory and calls `main` with no arguments, we take the function pointer to call, the initial memory, and the list of arguments as parameters. For the final state, we take not only the return value, but also the memory state when we exit the function.

Consequently, the compiler correctness proofs have to change. Currently, CompCert uses a downward simulation diagram [20, 2.1] for each pass from Clight, then, thanks to the fact that the CompCert assembly language is deterministic (up to input values given by the environment), CompCert composes all of them together before turning them to a single upward simulation which actually entails that the compiled code refines the source code.

In this work, we follow a similar approach: for each individual pass, we prove per-function semantics preservation in a downward simulation flavor. We do not, however, turn it into an upward simulation, because the whole layer refinement proof is based on downward simulation, which is in turn turned into an upward simulation at whole-machine contextual refinement thanks to the determinism (up to the environment) of LAsm(L).

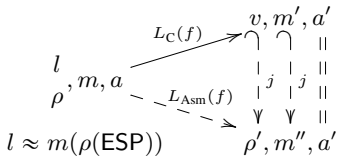
Memory state during compilation The main difference between CompCert and CompCertX lies in the memory given at the beginning of a function call.

In the whole-program setting, the initial state is the same across all languages, because it is uniquely determined by the global variables (which are preserved by compilation). On the other hand, in the middle of the execution when entering an arbitrary function, the memory in Clight is different from its assembly counterpart because CompCert introduces *memory transformations* such as memory injections or extensions [21, 5.4] to manage the callees' stack frames. This is actually advantageous for compilation of handling arguments and the return address.

For CompCertX, within the module being compiled, the same memory state mismatch also exists. At module entry, however, we cannot assume much about the memory state because it is given as a parameter to the semantics of each function in the module. In fact, this memory state is determined by the caller, so it may very well come from non-ClightX code (e.g., arbitrary assembly user code), thus we have to take the same memory as initial state across all the languages of CompCertX. It follows then that the arguments of the function already have to be present in the memory, following the calling convention imposed by the assembly language, even though ClightX does not read the arguments from memory.

Another difference between CompCert and CompCertX is the treatment of final memory states. In CompCert, only the return value of a program is observable at the end; the final memory state is not. By contrast, in CompCertX, the final memory state is passed back to the caller hence observable. Thus, it is necessary to account for memory transformations when relating the final states in the simulation diagrams.

Compilation refinement relation Finally, the per-function compiler correctness statement of CompCertX can be roughly summarized as this commutative diagram and formally defined below.



Definition 14. *Let L_C be a C layer interface, and L_{Asm} be an assembly layer interface. We say that L_C is simulated by L_{Asm} by compilation, written $L_C \leqslant^{comp} L_{Asm}$, if and only if, for any Γ, and for any execution $L_C(f)(l, m, a, v, m', a')$ of a primitive f of L_C for some list l of arguments and some return value v, from memory state m and abstract state a to m' and a', and for any register map ρ such that the following requirements hold:*

1. *the memory m contains the arguments l in the stack pointed to by $\rho(\mathsf{ESP})$*
2. EIP *points to the function f being called:* $\rho(\mathsf{EIP}) = (\Gamma(f), 0)$

Then, there is a primitive execution $L_{Asm}(f)(\rho, m, a, \rho', m'', a')$ and a memory injection j from m' to m'' preserving the addresses of m such that the following holds:

- *the values of callee-save registers in ρ are preserved in ρ';*
- $\rho'(\mathsf{EIP})$ *points to return address* $\rho(\mathsf{RA})$*;*
- *the return value contained in* $\rho'(\mathsf{EAX})$ *(for integers/pointers) or* $\rho'(\mathsf{FP0})$ *(for floating-points) is related to v by j;*

Theorem 2. *Let L be an assembly layer interface with all C-style primitives preserving memory transformations. Then, for any M:*

$$[\![M]\!]L \leqslant^{comp} [\![\text{CompCertX}(M)]\!]L$$

More details can be found in the companion technical report.

6.2 Linking compiled code with assembly code

Contrary to traditional separate compilation, we target compiling ClightX functions that may be called by LAsm assembly code. Since the caller may be arbitrary LAsm code, not necessarily well-behaved code written in or compiled from ClightX, we have to assume that the memory we are given follows LAsm layout. When reasoning about memory states that involve compiled code, we then have to accommodate memory injections introduced by the compiler.

During a whole-machine refinement proof, the two memory states of the overlay and the underlay are related with a simulation relation R. However, consider when the higher (LAsm) code calls an overlay primitive, that, in the underlay, is compiled from ClightX. Because during the per-primitive simulation proofs we ignored the effects of the compiler, the memory injection introduced by the compiler may become a source of discrepancy. That is why we encapsulate, in R, a memory injection between the higher memory state and the lower memory state. This injection is identity until the lower state calls a compiled ClightX function. Then, at every such call, the layer simulation relation R can "absorb" compilation refinement on its right-hand side:

Lemma 2. *If L' and L_C are C overlays and L_{Asm} is an assembly underlay, with $L' \leqslant_R L_C$ and $L_C \leqslant^{comp} L_{Asm}$, then $L' \leqslant_R L_{Asm}$.*

Proof. If R encapsulates a memory injection j_0, and compilation introduces a memory injection j, then, the simulation relation R will still hold with the composed memory injection $j \circ j_0$. □

Summary of the refinement proof with compilation and linking Finally, the outline of proving layer refinement $L_1 \vdash M : L_2$, where $M = \text{CompCertX}(M_\text{C}) \oplus M_\text{Asm}$ is the union of a compiled ClightX module and an LAsm module, is summarized in the following steps, also shown in Fig. 13:

1. Split the overlay L_2 into two layer interfaces $L_{2,\text{C}}$ and $L_{2,\text{Asm}}$ where $L_{2,\text{C}}$ is a C layer interface containing primitive specifications to be implemented by ClightX code (necessarily C-style) and $L_{2,\text{Asm}}$ is an assembly layer interface containing all other primitives (implemented in LAsm), so that $L_2 = L_{2,\text{C}} \oplus L_{2,\text{Asm}}$.
2. For each such part of the overlay, design an intermediate layer interface $L'_{1,\text{C}}$ and $L'_{1,\text{Asm}}$ with the same abstract state type as L_1 (see Section 4.3), and prove $L_{2,\text{C}} \leqslant_R L'_{1,\text{C}}$ and $L_{2,\text{Asm}} \leqslant_R L'_{1,\text{Asm}}$ independently of the implementation.
3. For both intermediate layer interfaces, prove that they are implemented by modules M_C and M_Asm on top of L_1 respectively, i.e. $L'_{1,\text{C}} \leqslant_{\mathbf{id}} [\![M_\text{C}]\!]L_1$ and $L'_{1,\text{Asm}} \leqslant_{\mathbf{id}} [\![M_\text{Asm}]\!]L_1$.
4. Then, compile M_C: $[\![M_\text{C}]\!]L_1 \leqslant^{\text{comp}} [\![\text{CompCertX}(M_\text{C})]\!]L_1$.

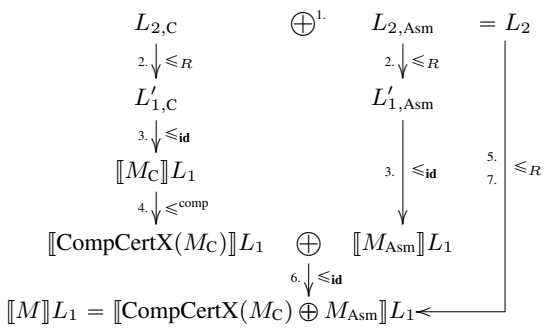


Figure 13. Proof steps of layer refinement $L_1 \vdash_R M : L_2$

5. Using LLE-TRANS and LLE-MON to combine 2. and 3., we have: $L_{2,\text{C}} \oplus L_{2,\text{Asm}} \leqslant_R L'_{1,\text{C}} \oplus L'_{1,\text{Asm}} \leqslant_{\mathbf{id}} [\![M_\text{C}]\!]L_1 \oplus [\![M_\text{Asm}]\!]L_1$
   On the C side (left of $\oplus$), Lemma 2 shows that $\leqslant_R$ absorbs $\leqslant^{\text{comp}}$. By 4.: $L_{2,\text{C}} \oplus L_{2,\text{Asm}} \leqslant_R [\![\text{CompCertX}(M_\text{C})]\!]L_1 \oplus [\![M_\text{Asm}]\!]L_1$
6. From the soundness of HCOMP (proof in TR [13]), and because $M = \text{CompCertX}(M_\text{C}) \oplus M_\text{Asm}$, we have:
   $$[\![\text{CompCertX}(M_\text{C})]\!]L_1 \oplus [\![M_\text{Asm}]\!]L_1 \leqslant_{\mathbf{id}} [\![M]\!]L_1$$
7. Finally, by combining 5. and 6., we have $L_{2,\text{C}} \oplus L_{2,\text{Asm}} \leqslant_R [\![M]\!]L_1$. Since $L_2 = L_{2,\text{C}} \oplus L_{2,\text{Asm}}$, by using LLE-UB-LEFT and LLE-COMM, we have: $L_2 \leqslant_R [\![M]\!]L_1 \leqslant_{\mathbf{id}} [\![M]\!]L_1 \oplus L_1 \leqslant_{\mathbf{id}} L_1 \oplus [\![M]\!]L_1$, thus we get $L_1 \vdash_R M : L_2$.

7. Case study: certified OS kernels

To demonstrate the power of our new languages and tools, we have applied our new layered approach to specify and verify four variants of mCertiKOS kernels in the Coq proof assistant. This section describes these kernels and the benefits of the approach.

The mCertiKOS base kernel is a simplified uniprocessor version of the CertiKOS kernel [12] designed for the 32 bit x86 architecture. It provides a multi-process environment for user-space applications using separate virtual address space, where the communications between different applications are established by message passing. The mCertiKOS-hyp kernel, built on top of the base kernel, is a realistic hypervisor kernel that can boot recent versions of unmodified Linux operating systems (Debian 6.0.6 and Ubuntu 12.04.2). The mCertiKOS-rz kernel extends the hypervisor supporting "ring 0" processes, hosting "certifiably safe" services and application programs inside the kernel address space. Finally, we strip the last kernel down to the mCertiKOS-emb kernel, removing virtualization, virtual memory, and user-space interrupt handling. This results in a minimal operating system suitable for embedded environments.

The layer structures of these kernels are shown in the top half of Fig. 14; each block in the top half represents a collection of sub-layers shown in the bottom half (as we zoom in on mCertiKOS-hyp).

mCertiKOS The layered approach is the key to our success in fully certifying a kernel. In Sec. 4.3, we have shown how to define getters and setters for abstract data types like those in Fig. 8, allowing higher layers to manipulate abstract states. Furthermore, layering is also crucial to certification of thread queues as discussed in Sec. 2. Instead of directly proving that a C linked-list implements a functional list, we insert an intermediate layer as shown in Fig. 1 to divide the difficult task into two steps.

These may look like mere proof techniques for enabling abstract states or reducing proof effort, but they echo the following mantra which makes our certification more efficient and scalable:

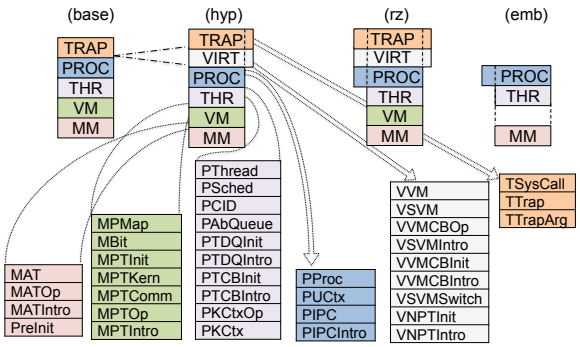


Figure 14. Various mCertiKOS layer structures. Layer short-hands: TRAP: interrupt handling; VIRT: virtualization; PROC: process management; THR: thread management; VM: virtual memory; MM: physical memory management.

> *Abstract in minimal steps, specify* full *behavior, and hide* all *underlying details.*

This is also how we prove the overall contextual correctness guarantees for all system calls and interrupt handlers. Fig. 15 shows the call graph of the page fault handler, including all functions called both directly and indirectly. Circles indicate functions, solid arrows mean primitive invocations, and faint dashed lines are primitives that are translated by all the layers they pass through.

Defined in TSysCall layer interface, the page fault handler makes use of proc_exit and proc_start, both defined in PProcd layer interface. Since the invocations of them are separated by other primitive calls, one may expect that the invariants need to be re-established or the effects of the in-between calls re-interpreted. Fortunately, as our mantra suggests, when the in-between layers translate the two primitives to TTrap layer interface, the behaviors of them are *fully* specified in terms of TTrap's abstract states, and the invariants of PProc layer interface are considered the underlying details and have *all* been hidden. This is especially important for calls like proc_exit to ikern_set which span over 20 layers with the abstract states so different that direct translation is not feasible.

Finally, kernel initialization is another difficult task that has been missing from other kernel verification projects. The traditional kernel initialization process is not compatible with *"specify full behavior and hide all underlying details."* For example, start_kernel in Linux kernel makes a sequence of calls to module initializations. mCertiKOS's initialization (see its call graph in Fig. 16) is a *chain* of calls to layer initializations; this pattern complies with the guideline that initializing one layer should hide the detail about initializing the lower layers. Without layering, the specifications of *all* functions will be populated with initialization flags for each module they depend on. This makes encapsulation harder and could also lead to a quadratic blowup in size and proving effort.

mCertiKOS-hyp The mCertiKOS-hyp kernel provides core primitives to build full-fledged user-level hypervisors by supporting one of the two popular hardware virtualization technologies – AMD SVM. The primitives include the operations for manipulating the virtual machine status, handling VMEXITs, starting or stopping a virtual machine, *etc*. The details of virtualization, e.g., the virtual machine control block and the nested page table, are hidden from the guest applications. The hypervisor functionalities are implemented in nine layers and then inserted in between process management and interrupt handling layers. The layered approach allows us to do so while (1) only modeling virtualization-specific structures when needed; (2) retaining primitives in the layer interface PProc by systematic lifting; and (3) adding new primitives (including a new initialization function) guaranteed not to interfere with existing primitives.

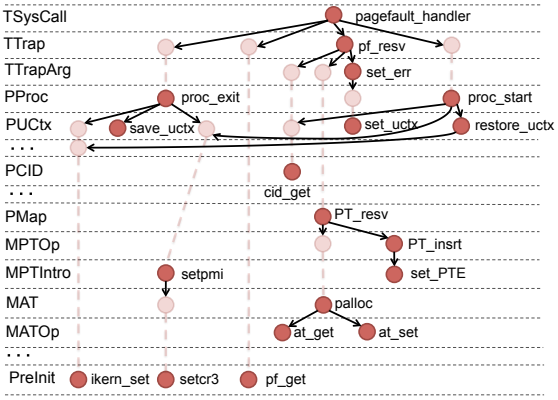


Figure 15. Call graph of the page fault handler

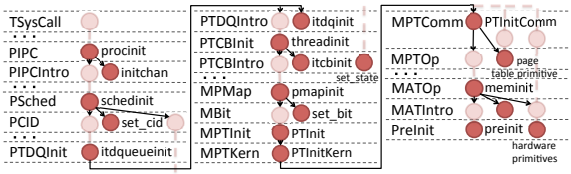


Figure 16. Call graph of mCertiKOS initializer

mCertiKOS-rz The mCertiKOS-rz kernel explores a different dimension—instead of adding intermediate layers, we augmented a few existing layers (in mCertiKOS-hyp) with support of ring 0 processes. The main modification is at PProc, where an additional kind of threads is defined. However, all the layers between PProc and TSysCall also need to be extended to expose the functionality as system calls. Thankfully, since all the new primitives are already described in deep specifications, lifting them to system calls only requires equality reasoning in Coq.

mCertiKOS-emb The mCertiKOS-emb kernel cuts features down to a bare minimum: it does not switch to user mode, hence does not require memory protection and does not provide system call interfaces. This requires *removing* features instead of adding them. Since the layered structure minimizes entanglements by eliminating unnecessary dependencies and code coupling, the removal process was relatively easy and straightforward. Moreover, removing the top 12 layers requires no additional specifications for those now top-level primitives—deep specifications are suitable for both internal reasonings and external descriptions. Thread and process management layers now sit directly on top of physical memory management; virtual memory is never enabled. The layers remain largely the same barring the removal of primitives mentioning page tables.

Evaluation and limitations The planning and development of mCertiKOS took 9.5 person months plus 2 person months on linking and code extraction. With the infrastructure in place, mCertiKOS-hyp only took 1.5 person months to finish, and mCertiKOS-rz and mCertiKOS-emb take half a person month each. The kernels are written, layer by layer, in LAsm and ClightX abstract syntaxes along with driver functions specifying how to compose (link) them. All of those are in Coq for the proofs to refer to. We utilize Coq's code extraction to get an OCaml program which contains CompCertX, the abstract syntax trees of the kernels, and the driver functions, which invoke CompCertX on pieces of ClightX code and generate the full assembly file. The output of the OCaml program is then fed to an assembler to produce the kernel executable.

With the device drivers (running as user processes) and a cooperative scheduler, most of the benchmarks in lmbench are under 2x slowdown running in mCertiKOS-hyp, well within expected over-

head. Ring 0 processes, not used in the above experiment, can easily lower the number as we measured one to two orders of magnitude reduction in the number of cycles needed to serve system calls.

Because the proof was originally developed directly in terms of abstract machines and program transformations, the current code base does not yet reflect the calculus presented in Sec. 3 in its entirety. Notably, vertical composition is done at the level of the whole-machine contextual refinements obtained by applying the soundness theorem to each individual abstraction layer.

Outside our verified kernels (mCertiKOS-hyp consists of about 3000 lines of C and assembly), there are 300 lines of C and 170 lines of x86 assembly code that are not verified yet: the preinit procedure, the ELF loader used by user process creation, and functions such as memcpy which currently cannot be verified because of a limitation arising from the CompCert memory model. Device drivers are not verified because LAsm lacks device models for expressing the correctness statement. Finally, the CompCert assembler for converting LAsm into machine code remains unverified.

8. Related work

Hoare-style program verification Hoare logic [14] and its modern variants [32, 2, 26] were introduced to prove strong (partial or total) correctness properties of programs annotated with pre- and postconditions. A total-correctness Hoare triple $[P]C[Q]$ often means a refinement between the implementation C and the specification $[P, Q]$: given any state S, if the precondition $P(S)$ holds, then the command C can run safely and terminate with a state that satisfies Q. Though not often done, it is also possible to introduce auxiliary/ghost states to serve as "abstract states" and prove that a program implements a specification via a simulation.

Our layer language can be viewed as a novel way of imposing a module system over Hoare-style program verification. We insist on using interfaces with deep specifications and we address the "conflicting abstract states" problem mentioned in Sec. 2. Traditional program verification does not always use deep specification (for pre- and post-conditions) so the module interfaces (e.g., $[P, Q]$) may allow some safe but unwanted behaviors. Such gap is fine if the goal is to just prove safety (as in static type-checking), but if we want to prove the strong contextual correctness property across module boundaries, it is important that each interface accurately describes the functionality and scope of the underlying implementation.

In addition to the obvious benefits on compositionality, our layered approach also enables a new powerful way of combining programming- and specification languages in a single setting. Each layer interface enables a new programming language at a specific abstraction level, which is then used to implement layers at even higher levels. As we move up the layer hierarchy, our programming language gets closer and closer to the specification language—it can call primitives at higher abstraction levels but it still supports general-purpose programming (e.g., in ClightX).

Interestingly, we did not need to introduce any program logic to verify our OS kernel code. Instead, we verify it directly using the ClightX (or LAsm) language semantics (which is already conveniently parameterized over a layer interface). In fact, unlike Hoare logic which shows that a program (e.g., C) refines a specification (e.g., $[P, Q]$), we instead show there is a downward simulation from the specification to the program. As in CompCert, we found this easier to prove and we can do this because both our specification and language semantics are deterministic relative to external events.

Stepwise program refinement Dijkstra [9] proposed to "realize" a complex program by decomposing it into a hierarchy of linearly ordered "abstract machines." Based on this idea, the PSOS team at SRI [27] developed the Hierarchical Development Methodology (HDM) and applied HDM to design and specify an OS using 20 hierarchically organized modules. HDM was difficult to be rigorously applied in practice, probably because of the lack of powerful specification and proof tools. In this paper, we advance the HDM paradigm by using a new formal layer language to connect multiple layers and by implementing all certified layers and proofs in a modern proof assistant. We also pursued decomposition more aggressively since it made our verification task much easier.

Morgan's refinement calculus [25] is a formalized approach to Dijkstra's stepwise refinement. Using this calculus, a high-level specification can be refined through a series of correctness-preserving transformations and eventually turned into an efficient executable. Our work imposes a new layer language to enhance compositional reasoning. We use ClightX (or LAsm) and the Coq logic as our "refinement" language, and use a certified layer (with deep specification) to represent each such correctness-preserving transformation. All our ClightX and LAsm instances have executable semantics and can be compiled and linked using our new CompCertX compiler.

Separate compilation for CompCert Compositional compiler correctness is an extremely challenging problem [3, 15], especially when it involves an open compiler with multiple languages [29]. In the context of CompCert, a recent proposal [4] aims to tackle the full Clight language but it has not been fully implemented in the CompCert compiler. While our CompCertX compiler proves a stronger correctness theorem for each ClightX layer, the ClightX language is subtly different from the original full-featured Clight language. Within each ClightX layer, all locally allocated memory blocks (e.g., stack frames) cannot be updated by functions defined in another layer. This means that ClightX does not support the same general "stack-allocated data structures" as in Clight. This is fine for our OS kernels since they do not allocate any data structures on stack, but it means that CompCertX can not be regarded as a full featured separate compiler for CompCert.

OS kernel verification The seL4 team [17] were the first to build a proof of functional correctness for a realistic microkernel. The seL4 work is impressive in that all the proofs were done inside a modern mechanized proof assistant. They have shown that the behaviors of 7500 lines of their C code always follow an abstract specification of their kernel. To make verification easier, they introduced an intermediate executable specification to hide C specifics. Both their abstract and executable specifications are "monolithic" as they are not divided into layers to support abstraction among different kernel modules. These kernel interdependencies led to more complex invariants which may explain why their effort took 11 person years.

The initial seL4 effort was done completely at the C level so it does not support many assembly level features such as address translation. This also made verification of assembly code and kernel initialization difficult (1200 lines of C and 500 lines of assembly are still unverified). It is also unclear how to use their verified kernel to reason about user-level programs since they would be running in a different address space. Our certified kernels, on the other hand, directly model assembly-level machines that support all kernel/user and host/guest programs. Memory access to a user-level address space must go through a page table, and memory access in a guest virtual machine must go through a nested page table. We thus had no problem verifying our kernel initialization or assembly code.

Modular verification of low-level code Vaynberg and Shao [36] also used a layered approach to verify a small virtual memory manager. Their layers are not linearly ordered; instead, their seven abstract machines form a DAG with potential upcalls (i.e., calls from a lower layer to upper ones). As a result, their initialization function (an upcall) was much harder to verify. Their refinement proofs between layers are insensitive to termination, from which they can only prove partial correctness but not the strong contextual correctness property which we prove in our current work.

Feng *et al.* [11] developed OCAP, an open framework for linking components verified in different domain-specific program logics. They verified a thread library with hardware interrupts and preemption [10] using a variant of concurrent separation logic [28]. They decomposed the thread implementation into a sequential layer (with interrupts disabled) and a concurrent layer (with interrupts enabled). Chlipala [8] developed Bedrock, an automated Coq library to support verified low-level programming. All these systems aimed to prove *partial correctness* only, so they are quite different from the layered simulation proofs given in this paper.

9. Conclusions

Abstraction layers are key techniques used in building large-scale computer software and hardware. In this paper, we have presented a novel language-based account of abstraction layers and shown that they are particularly suitable for supporting abstraction over deep specifications, which is essential for compositional verification of strong correctness properties. We have designed a new layer language and imposed it on two different core languages (ClightX and LAsm). We have also built a verified compiler from ClightX to LAsm. By aggressively decomposing each complex abstraction into smaller abstraction steps, we have successfully developed several certified OS kernels that prove deeper properties (contextual correctness), contain smaller trusted computing bases (all code verified at the assembly level), require significantly less effort (3000 lines of C and assembly code proved in less than 1 person year), and demonstrate strong support for extensibility (layers are heavily reused in different certified kernels). We expect that both deep specifications and certified abstraction layers will become critical technologies and important building blocks for developing large-scale certified system infrastructures in the future.

Acknowledgments We thank Quentin Carbonneaux, David Costanzo, Rance DeLong, Xinyu Feng, Bryan Ford, Liang Gu, Jan Hoffmann, Hongjin Liang, Joshua Lockerman, Peter Neumann, David Pichardie, members of the CertiKOS team at Yale, and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, NSF grants 1065451 and 0915888, and ONR Grant N00014-12-1-0478. It is also supported in part by China Scholarship Council and National Natural Science Foundation of China (NSFC grants 61229201 and 61202052). Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] C. Y. Baldwin and K. B. Clark. *Design Rules: Volume 1, The Power of Modularity*. MIT Press, March 2000.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. 4th Symp on Formal Methods for Components and Objects*, 2005.

[3] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP'09*, pages 97–108, 2009.

[4] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *ESOP'14*, pages 107–127, 2014.

[5] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3):263–288, 2009.

[6] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI'14*.

[7] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL'10*, pages 57–69, 2010.

[8] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245, 2011.

[9] E. W. Dijkstra. Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press, 1972.

[10] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI'08*, pages 170–182, June 2008.

[11] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Combining domain-specific and foundational logics to verify complete software systems. In *VSTTE'08*, pages 54–69, 2008.

[12] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: a certified kernel for secure cloud computing. In *APSys '11*, 2011.

[13] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. Yale Univ. Technical Report YALEU/DCS/TR-1500; `http://flint.cs.yale.edu/publications/dscal.html`, Oct. 2014.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[15] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL'12*, pages 59–72.

[16] D. Jackson. *Software abstractions: logic, languages, and analysis*. The MIT Press, 2012.

[17] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, et al. seL4: Formal verification of an OS kernel. In *SOSP'09*, pages 207–220, October 2009.

[18] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[19] X. Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2014.

[20] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[21] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformation. *J. Automated Reasoning*, 41(1):1–31, 2008.

[22] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[24] J. C. Mitchell. Representation independence and data abstraction. In *POPL'86*, pages 263–276, January 1986.

[25] C. C. Morgan. *Programming from specifications, 2nd Edition*. Prentice-Hall, 1994.

[26] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In *ICFP'06*, pages 62–73, Sept. 2006.

[27] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: its system, its applications, and proofs. Technical Report CSL-116, SRI, May 1980.

[28] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04*, pages 49–67, 2004.

[29] J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In *ESOP'14*, pages 128–148, 2014.

[30] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[31] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[32] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74, 2002.

[33] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), 2013.

[34] M. Spivey. *The Z Notation: A reference manual*. Prentice Hall, 1992.

[35] The Coq development team. The Coq proof assistant. `http://coq.inria.fr`, 1999 – 2014.

[36] A. Vaynberg and Z. Shao. Compositional verification of a baby virtual memory manager. In *CPP'12*, pages 143–159, Dec 2012.

Automatic Static Cost Analysis for Parallel Programs

Jan Hoffmann and Zhong Shao

Yale University

Abstract. Static analysis of the evaluation cost of programs is an extensively studied problem that has many important applications. However, most automatic methods for static cost analysis are limited to sequential evaluation while programs are increasingly evaluated on modern multicore and multiprocessor hardware. This article introduces the first automatic analysis for deriving bounds on the worst-case evaluation cost of parallel first-order functional programs. The analysis is performed by a novel type system for amortized resource analysis. The main innovation is a technique that separates the reasoning about sizes of data structures and evaluation cost within the same framework. The cost semantics of parallel programs is based on call-by-value evaluation and the standard cost measures *work* and *depth*. A soundness proof of the type system establishes the correctness of the derived cost bounds with respect to the cost semantics. The derived bounds are multivariate resource polynomials which depend on the sizes of the arguments of a function. Type inference can be reduced to linear programming and is fully automatic. A prototype implementation of the analysis system has been developed to experimentally evaluate the effectiveness of the approach. The experiments show that the analysis infers bounds for realistic example programs such as quick sort for lists of lists, matrix multiplication, and an implementation of sets with lists. The derived bounds are often asymptotically tight and the constant factors are close to the optimal ones.

Keywords: Functional Programming, Static Analysis, Resource Consumption, Amortized Analysis

1 Introduction

Static analysis of the resource cost of programs is a classical subject of computer science. Recently, there has been an increased interest in formally proving cost bounds since they are essential in the verification of safety-critical real-time and embedded systems.

For sequential functional programs there exist many automatic and semi-automatic analysis systems that can statically infer cost bounds. Most of them are based on sized types [1], recurrence relations [2], and amortized resource analysis [3, 4]. The goal of these systems is to automatically compute easily-understood arithmetic expressions in the sizes of the inputs of a program that bound resource cost such as time or space usage. Even though an automatic

computation of cost bounds is undecidable in general, novel analysis techniques are able to efficiently compute tight time bounds for many non-trivial programs [5–9].

For functional programs that are evaluated in parallel, on the other hand, no such analysis system exists to support programmers with computer-aided derivation of cost bounds. In particular, there are no type systems that derive cost bounds for parallel programs. This is unsatisfying because parallel evaluation is becoming increasingly important on modern hardware and referential transparency makes functional programs ideal for parallel evaluation.

This article introduces an automatic type-based resource analysis for deriving cost bounds for parallel first-order functional programs. Automatic cost analysis for sequential programs is already challenging and it might seem to be a long shot to develop an analysis for parallel evaluation that takes into account low-level features of the underlying hardware such as the number of processors. Fortunately, it has been shown [10, 11] that the cost of parallel functional programs can be analyzed in two steps. First, we derive cost bounds at a high abstraction level where we assume to have an unlimited number of processors at our disposal. Second, we prove once and for all how the cost on the high abstraction level relates to the actual cost on a specific system with limited resources.

In this work, we derive bounds on an abstract cost model that consists of the *work* and the *depth* of an evaluation of a program [10]. Work measures the evaluation time of sequential evaluation and depth measures the evaluation time of parallel evaluation assuming an unlimited number of processors. It is well-known [12] that a program that evaluates to a value using work w and depth d can be evaluated on a shared-memory multiprocessor (SMP) system with p processors in time $O(\max(w/p, d))$ (see Section 2.3). The mechanism that is used to prove this result is comparable to a scheduler in an operating system.

A novelty in the cost semantics in this paper is the definition of work and depth for terminating and non-terminating evaluations. Intuitively, the non-deterministic big-step evaluation judgement that is defined in Section 2 expresses that *there is a (possibly partial) evaluation with work n and depth m*. This statement is used to prove that a typing derivation for bounds on the depth or for bounds on the work ensures termination.

Technically, the analysis computes two separate typing derivations, one for the work and one for the depth. To derive a bound on the work, we use multivariate amortized resource analysis for sequential programs [13]. To derive a bound on the depth, we develop a novel multivariate amortized resource analysis for programs that are evaluated in parallel. The main challenge in the design of this novel parallel analysis is to ensure the same high compositionality as in the sequential analysis. The design and implementation of this novel analysis for bounds on the depth of evaluations is the main contribution of our work. The technical innovation that enables compositionality is an analysis method that separates the static tracking of size changes of data structures from the cost analysis while using the same framework. We envision that this technique will find further applications in the analysis of other non-additive cost such as stack-space usage and recursion depth.

We describe the new type analysis for parallel evaluation for a simple first-order language with lists, pairs, pattern matching, and sequential and parallel composition. This is already sufficient to study the cost analysis of parallel programs. However, we implemented the analysis system in Resource Aware ML (RAML), which also includes other inductive data types and conditionals [14]. To demonstrate the universality of the approach, we also implemented NESL's [15] parallel list comprehensions as a primitive in RAML (see Section 6). Similarly, we can define other parallel sequence operations of NESL as primitives and correctly specify their work and depth. RAML is currently extended to include higher-order functions, arrays, and user-defined inductive types. This work is orthogonal to the treatment of parallel evaluation.

To evaluate the practicability of the proposed technique, we performed an experimental evaluation of the analysis using the prototype implementation in RAML. Note that the analysis computes worst-case bounds instead of average-case bounds and that the asymptotic behavior of many of the classic examples of Blelloch et al. [10] does not differ in parallel and sequential evaluations. For instance, the depth and work of quick sort are both quadratic in the worst-case. Therefore, we focus on examples that actually have asymptotically different bounds for the work and depth. This includes quick sort for lists of lists in which the comparisons of the inner lists can be performed in parallel, matrix multiplication where matrices are lists of lists, a function that computes the maximal weight of a (continuous) sublist of an integer list, and the standard operations for sets that are implemented as lists. The experimental evaluation can be easily reproduced and extended: RAML and the example programs are publicly available for download and through an user-friendly online interface [16].

In summary we make the following contributions.

1. We introduce the first automatic static analysis for deriving bounds on the depth of parallel functional programs. Being based on multivariate resource polynomials and type-based amortized analysis, the analysis is compositional. The computed type derivations are easily-checkable bound certificates.
2. We prove the soundness of the type-based amortized analysis with respect to an operational big-step semantics that models the work and depth of terminating and non-terminating programs. This allows us to prove that work and depth bounds ensure termination. Our inductively defined big-step semantics is an interesting alternative to coinductive big-step semantics.
3. We implemented the proposed analysis in RAML, a first-order functional language. In addition to the language constructs like lists and pairs that are formally described in this article, the implementation includes binary trees, natural numbers, tuples, Booleans, and NESL's parallel list comprehensions.
4. We evaluated the practicability of the implemented analysis by performing reproducible experiments with typical example programs. Our results show that the analysis is efficient and works for a wide range of examples. The derived bounds are usually asymptotically tight if the tight bound is expressible as a resource polynomial.

The full version of this article [17] contains additional explanations, lemmas, and details of the technical development.

2 Cost Semantics for Parallel Programs

In this section, we introduce a first-order functional language with parallel and sequential composition. We then define a big-step operational semantics that formalizes the cost measures *work* and *depth* for terminating and non-terminating evaluations. Finally, we prove properties of the cost semantics and discuss the relation of work and depth to the run time on hardware with finite resources.

2.1 Expressions and Programs

Expressions are given in let-normal form. This means that term formers are applied to variables only when this does not restrict the expressivity of the language. Expressions are formed by integers, variables, function applications, lists, pairs, pattern matching, and sequential and parallel composition.

$$\begin{aligned} e, e_1, e_2 ::= {} & n \mid x \mid f(x) \mid (x_1, x_2) \mid\mid \mathsf{match}\, x \,\mathsf{with}\, (x_1, x_2) \Rightarrow e \\ & \mid \mathsf{nil} \mid \mathsf{cons}(x_1, x_2) \mid \mathsf{match}\, x \,\mathsf{with}\, \langle \mathsf{nil} \Rightarrow e_1 \mid \mathsf{cons}(x_1, x_2) \Rightarrow e_2 \rangle \\ & \mid \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \mid \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e \end{aligned}$$

The parallel composition $\mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e$ is used to evaluate e_1 and e_2 in parallel and bind the resulting values to the names x_1 and x_2 for use in e.

In the prototype, we have implemented other inductive types such as trees, natural numbers, and tuples. Additionally, there are operations for primitive types such as Booleans and integers, and NESL's parallel list comprehensions [15]. Expressions are also transformed automatically into let normal form before the analysis. In the examples in this paper, we use the syntax of our prototype implementation to improve readability.

In the following, we define a standard type system for expressions and programs. Data types A, B and function types F are defined as follows.

$$A, B ::= \mathrm{int} \mid L(A) \mid A * B \qquad\qquad F ::= A \to B$$

Let $\mathcal{A}$ be the set of data types and let $\mathcal{F}$ be the set of function types. A signature $\Sigma : \mathrm{FID} \rightharpoonup \mathcal{F}$ is a partial finite mapping from function identifiers to function types. A context is a partial finite mapping $\Gamma : \mathit{Var} \rightharpoonup \mathcal{A}$ from variable identifiers to data types. A simple type judgement $\Sigma; \Gamma \vdash e : A$ states that the expression e has type A in the context Γ under the signature Σ. The definition of typing rules for this judgement is standard and we omit the rules.

A *(well-typed) program* consists of a signature Σ and a family $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ of expressions e_f with a distinguished variable identifier y_f such that $\Sigma; y_f{:}A \vdash e_f{:}B$ if $\Sigma(f) = A \to B$.

2.2 Big-Step Operational Semantics

We now formalize the resource cost of evaluating programs with a big-step operational semantics. The focus of this paper is on time complexity and we only define the cost measures *work* and *depth*. Intuitively, the work measures the time that is needed in a sequential evaluation. The depth measures the time that is needed in a parallel evaluation. In the semantics, time is parameterized by a metric that assigns a non-negative cost to each evaluation step.

$$\frac{V, H \overset{M}{\vdash} e_1 \Downarrow \circ \mid (w, d)}{V, H \overset{M}{\vdash} \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow \circ \mid (M^{\mathsf{let}}{+}w, M^{\mathsf{let}}{+}d)}\,(\text{E:LET1}) \qquad \frac{}{V, H \overset{M}{\vdash} e \Downarrow \circ \mid (0,0)}\,(\text{E:ABORT})$$

$$\frac{V, H \overset{M}{\vdash} e_1 \Downarrow (\ell, H') \mid (w_1, d_1) \qquad V[x \mapsto \ell], H' \overset{M}{\vdash} e_2 \Downarrow \rho \mid (w_2, d_2)}{V, H \overset{M}{\vdash} \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \Downarrow \rho \mid (M^{\mathsf{let}}{+}w_1{+}w_2, M^{\mathsf{let}}{+}d_1{+}d_2)}\,(\text{E:LET2})$$

$$\frac{V, H \overset{M}{\vdash} e_1 \Downarrow \rho_1 \mid (w_1, d_1) \qquad V, H \overset{M}{\vdash} e_2 \Downarrow \rho_2 \mid (w_2, d_2) \qquad \rho_1{=}\circ \vee \rho_2{=}\circ}{V, H \overset{M}{\vdash} \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e \Downarrow \circ \mid (M^{\mathsf{Par}}{+}w_1{+}w_2, M^{\mathsf{Par}}{+}\max(d_1, d_2))}\,(\text{E:PAR1})$$

$$\frac{\begin{array}{l}V, H \overset{M}{\vdash} e_1 \Downarrow (\ell_1, H_1) \mid (w_1, d_1) \qquad (w', d'){=}(M^{\mathsf{Par}}{+}w_1{+}w_2{+}w, M^{\mathsf{Par}}{+}\max(d_1, d_2){+}d) \\ V, H \overset{M}{\vdash} e_2 \Downarrow (\ell_2, H_2) \mid (w_2, d_2) \qquad V[x_1{\mapsto}\ell_1, x_2{\mapsto}\ell_2], H_1 \uplus H_2 \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)\end{array}}{V, H' \overset{M}{\vdash} \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e \Downarrow (\ell, H') \mid (w', d')}\,(\text{E:PAR2})$$

Fig. 1. Interesting rules of the operational big-step semantics.

Motivation. A distinctive feature of our big-step semantics is that it models terminating, failing, and diverging evaluations by inductively describing finite subtrees of (possibly infinite) evaluation trees. By using an inductive judgement for diverging and terminating computations while avoiding intermediate states, it combines the advantages of big-step and small-step semantics. This has two benefits compared to standard big-step semantics. First, we can model the resource consumption of diverging programs and prove that bounds hold for terminating and diverging programs. (In some cost metrics, diverging computations can have finite cost.) Second, for a cost metric in which all diverging computations have infinite cost we are able to show that bounds imply termination.

Note that we cannot achieve this by step-indexing a standard big-step semantics. The available alternatives to our approach are small-step semantics and coinductive big-step semantics. However, it is unclear how to prove the soundness of our type system with respect to these semantics. Small-step semantics is difficult to use because our type-system models an intentional property that goes beyond the classic type preservation: After performing a step, we have to obtain a refined typing that corresponds to a (possibly) smaller bound. Coinductive derivations are hard to relate to type derivations because type derivations are defined inductively.

Our inductive big-step semantics can not only be used to formalize resource cost of diverging computations but also for other effects such as event traces. It is therefore an interesting alternative to recently proposed coinductive operational big-step semantics [18].

Semantic Judgements. We formulate the big-step semantics with respect to a stack and a heap. Let *Loc* be an infinite set of *locations* modeling memory addresses on a heap. A value $v ::= n \mid (\ell_1, \ell_2) \mid (\mathsf{cons}, \ell_1, \ell_2) \mid \mathsf{nil} \in \mathit{Val}$ is either an integer $n \in \mathbb{Z}$, a pair of locations (ℓ_1, ℓ_2), a node $(\mathsf{cons}, \ell_1, \ell_2)$ of a list, or nil.

A *heap* is a finite partial mapping $H : \mathit{Loc} \rightharpoonup \mathit{Val}$ that maps locations to values. A *stack* is a finite partial mapping $V : \mathit{Var} \rightharpoonup \mathit{Loc}$ from variable identifiers

to locations. Thus we have boxed values. It is not important for the analysis whether values are boxed.

Figure 1 contains a compilation of the big-step evaluation rules (the full version contains all rules). They are formulated with respect to a resource metric M. They define the evaluation judgment

$$V, H \overset{M}{\vdash} e \Downarrow \rho \mid (w, d) \qquad \text{where} \qquad \rho ::= (\ell, H) \mid \circ \,.$$

It expresses the following. In a fixed program $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$, if the stack V and the initial heap H are given then the expression e evaluates to ρ. Under the metric M, the work of the evaluation of e is w and the depth of the evaluation is d. Unlike standard big-step operational semantics, ρ can be either a pair of a location and a new heap, or $\circ$ (pronounced *busy*) indicating that the evaluation is not finished yet.

A resource metric $M : K \to \mathbb{Q}_0^+$ defines the resource consumption in each evaluation step of the big-step semantics with a non-negative rational number. We write M^k for $M(k)$.

An intuition for the judgement $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (w, d)$ is that there is a partial evaluation of e that runs without failure, has work w and depth d, and has not yet reached a value. This is similar to a small-step judgement.

Rules. For a heap H, we write $H, \ell \mapsto v$ to express that $\ell \notin \mathrm{dom}(H)$ and to denote the heap H' such that $H'(x) = H(x)$ if $x \in \mathrm{dom}(H)$ and $H'(\ell) = v$. In the rule E:Par2, we write $H_1 \uplus H_2$ to indicate that H_1 and H_2 agree on the values of locations in $\mathrm{dom}(H_1) \cap \mathrm{dom}(H_2)$ and to a combined heap H with $\mathrm{dom}(H) = \mathrm{dom}(H_1) \cup \mathrm{dom}(H_2)$. We assume that the locations that are allocated in parallel evaluations are disjoint. That is easily achievable in an implementation.

The most interesting rules of the semantics are E:Abort, and the rules for sequential and parallel composition. They allow us to approximate infinite evaluation trees for non-terminating evaluations with finite subtrees. The rule E:Abort states that we can partially evaluate every expression by doing zero steps. The work w and depth d are then both zero (i.e., $w = d = 0$).

To obtain an evaluation judgement for a sequential composition $\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ we have two options. We can use the rule E:Let1 to partially evaluate e_1 using work w and depth d. Alternatively, we can use the rule E:Let2 to evaluate e_1 until we obtain a location and a heap (ℓ, H') using work w_1 and depth d_1. Then we evaluate e_2 using work w_2 and depth d_2. The total work and depth is then given by $M^{\mathsf{let}}{+}w_1{+}w_2$ and $M^{\mathsf{let}}{+}d_1{+}d_2$, respectively.

Similarly, we can derive evaluation judgements for a parallel composition $\mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e$ using the rules E:Par1 and E:Par2. In the rule E:Par1, we partially evaluate e_1 or e_2 with evaluation cost (w_1, d_1) and (w_2, d_2). The total work is then $M^{\mathsf{Par}}{+}w_1{+}w_2$ (the cost for the evaluation of the parallel binding plus the cost for the sequential evaluation of e_1 and e_2). The total depth is $M^{\mathsf{Par}} + \max(d_1, d_2)$ (the cost for the evaluation of the binding plus the maximum of the cost of the depths of e_1 and e_2). The rule E:Par2 handles the case in which e_1 and e_2 are fully evaluated. It is similar to E:Let2 and the cost of the evaluation of the expression e is added to both the cost and the depth since e is evaluated after e_1 and e_2.

2.3 Properties of the Cost-Semantics

The main theorem of this section states that the resource cost of a partial evaluation is less than or equal to the cost of an evaluation of the same expression that terminates.

Theorem 1. *If* $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ *and* $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (w', d')$ *then* $w' \leqslant w$ *and* $d' \leqslant d$.

Theorem 1 can be proved by a straightforward induction on the derivation of the judgement $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$.

Provably Efficient Implementations. While work is a realistic cost-model for the sequential execution of programs, depth is not a realistic cost-model for parallel execution. The main reason is that it assumes that an infinite number of processors can be used for parallel evaluation. However, it has been shown [10] that work and depth are closely related to the evaluation time on more realistic abstract machines.

For example, *Brent's Theorem* [12] provides an asymptotic bound on the number of execution steps on the shared-memory multiprocessor (SMP) machine. It states that if $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ then e can be evaluated on a p-processor SMP machine in time $O(\max(w/p, d))$. An SMP machine has a fixed number p of processes and provides constant-time access to a shared memory. The proof of Brent's Theorem can be seen as the description of a so-called *provably efficient implementation*, that is, an implementation for which we can establish an asymptotic bound that depends on the number of processors.

Classically, we are especially interested in non-asymptotic bounds in resource analysis. It would thus be interesting to develop a non-asymptotic version of Brent's Theorem for a specific architecture using more refined models of concurrency [11]. However, such a development is not in the scope of this article.

Well-Formed Environments and Type Soundness. For each data type A we inductively define a set $[\![A]\!]$ of values of type A. Lists are interpreted as lists and pairs are interpreted as pairs.

$$[\![\text{int}]\!] = \mathbb{Z} \qquad [\![A * B]\!] = [\![A]\!] \times [\![B]\!]$$
$$[\![L(A)]\!] = \{[a_1, \ldots, a_n] \mid n \in \mathbb{N}, a_i \in [\![A]\!]\}$$

If H is a heap, ℓ is a location, A is a data type, and $a \in [\![A]\!]$ then we write $H \vDash \ell \mapsto a : A$ to mean that ℓ defines the semantic value $a \in [\![A]\!]$ when pointers are followed in H in the obvious way. The judgment is formally defined in the full version of the article.

We write $H \vDash \ell : A$ to indicate that there exists a, necessarily unique, semantic value $a \in [\![A]\!]$ so that $H \vDash \ell \mapsto a : A$. A stack V and a heap H are *well-formed* with respect to a context Γ if $H \vDash V(x) : \Gamma(x)$ holds for every $x \in \text{dom}(\Gamma)$. We then write $H \vDash V : \Gamma$.

Simple Metrics and Progress. In the reminder of this section, we prove a property of the evaluation judgement under a simple metric. A *simple metric* M assigns the value 1 to every resource constant, that is, $M(x) = 1$ for every $x \in K$. With a simple metric, work counts the number of evaluation steps.

Theorem 2 states that, in a well-formed environment, well-typed expressions either evaluate to a value or the evaluation uses unbounded work and depth.

Theorem 2 (Progress). *Let M be a simple metric, $\Sigma; \Gamma \vdash e : B$, and $H \models V : \Gamma$. Then $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ for some $w, d \in \mathbb{N}$ or for every $n \in \mathbb{N}$ there exist $x, y \in \mathbb{N}$ such that $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (x, n)$ and $V, H \overset{M}{\vdash} e \Downarrow \circ \mid (n, y)$.*

A direct consequence of Theorem 2 is that bounds on the depth of programs under a simple metric ensure termination.

3 Amortized Analysis and Parallel Programs

In this section, we give a short introduction into amortized resource analysis for sequential programs (for bounding the work) and then informally describe the main contribution of the article: a multivariate amortized resource analysis for parallel programs (for bounding the depth).

Amortized Resource Analysis. Amortized resource analysis is a type-based technique for deriving upper bounds on the resource cost of programs [3]. The advantages of amortized resource analysis are compositionality and efficient type inference that is based on linear programming. The idea is that types are decorated with resource annotations that describe a potential function. Such a potential function maps the sizes of typed data structures to a non-negative rational number. The typing rules ensure that the potential defined by a typing context is sufficient to pay for the evaluation cost of the expression that is typed under this context and for the potential of the result of the evaluation.

The basic idea of amortized analysis is best explained by example. Consider the function $\mathsf{mult} : \mathrm{int} * \mathsf{L}(\mathrm{int}) \to \mathsf{L}(\mathrm{int})$ that takes an integer and an integer list and multiplies each element of the list with the integer.

```
mult(x,ys) = match ys with | nil  → nil
                           | (y::ys') → x*y::mult(x,ys')
```

For simplicity, we assume a metric M^* that only counts the number of multiplications performed in an evaluation in this section. Then $V, H \overset{M^*}{\vdash} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) \Downarrow (\ell, H') \mid (n, n)$ for a well-formed stack V and heap H in which ys points to a list of length n. In short, the work and depth of the evaluation of $\mathsf{mult}(\mathsf{x}, \mathsf{ys})$ is $|\mathsf{ys}|$.

To obtain a bound on the work in type-based amortized resource analysis, we derive a type of the following form.

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \overset{M^*}{\vdash} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) : (L(\mathrm{int}), Q')$$

Here Q and Q' are *coefficients* of multivariate resource polynomials $p_Q : [\![\mathrm{int} * L(\mathrm{int})]\!] \to \mathbb{Q}_0^+$ and $p_{Q'} : [\![L(\mathrm{int})]\!] \to \mathbb{Q}_0^+$ that map semantic values to non-negative rational numbers. The rules of the type system ensure that for every evaluation context (V, H) that maps x to a number m and ys to a list a, the potential $p_Q(m, a)$ is sufficient to cover the evaluation cost of $\mathsf{mult}(\mathsf{x}, \mathsf{ys})$ and the potential $p_{Q'}(a')$ of the returned list a'. More formally, we have $p_Q(m, a) \geqslant w + p_{Q'}(a')$ if $V, H \overset{M^*}{\vdash} \mathsf{mult}(\mathsf{x}, \mathsf{ys}) \Downarrow (\ell, H') \mid (w, d)$ and ℓ points to the list a' in H'.

In our type system we can for instance derive coefficients Q and Q' that represent the potential functions

$$p_Q(n,a) = |a| \qquad \text{and} \qquad p_{Q'}(a) = 0\ .$$

The intuitive meaning is that we must have the potential $|\mathsf{ys}|$ available when evaluating $\mathsf{mult}(\mathsf{x},\mathsf{ys})$. During the evaluation, the potential is used to pay for the evaluation cost and we have no potential left after the evaluation.

To enable compositionality, we also have to be able to pass potential to the result of an evaluation. Another possible instantiation of Q and Q' would for example result in the following potential.

$$p_Q(n,a) = 2{\cdot}|a| \qquad \text{and} \qquad p_{Q'}(a) = |a|$$

The resulting typing can be read as follows. To evaluate $\mathsf{mult}(\mathsf{x},\mathsf{ys})$ we need the potential $2|\mathsf{ys}|$ to pay for the cost of the evaluation. After the evaluation there is the potential $|\mathsf{mult}(\mathsf{x},\mathsf{ys})|$ left to pay for future cost in a surrounding program. Such an instantiation would be needed to type the inner function application in the expression $\mathsf{mult}(\mathsf{x},\mathsf{mult}(\mathsf{z},\mathsf{ys}))$.

Technically, the coefficients Q and Q' are families that are indexed by sets of base polynomials. The set of base polynomials is determined by the type of the corresponding data. For the type $\mathrm{int} * L(\mathrm{int})$, we have for example $Q = \{q_{(*,[])}, q_{(*,[*])}, q_{(*,[*,*])}, \ldots\}$ and $p_Q(n,a) = q_{(*,[])} + q_{(*,[*])}{\cdot}|a| + q_{(*,[*,*])}{\cdot}\binom{|a|}{2} + \ldots$. This allows us to express multivariate functions such as $m \cdot n$.

The rules of our type system show how to describe the valid instantiations of the coefficients Q and Q' with a set of linear inequalities. As a result, we can use linear programming to infer resource bounds efficiently.

A more in-depth discussion can be found in the literature [3, 19, 7].

Sequential Composition. In a sequential composition $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$, the initial potential, defined by a context and a corresponding annotation (Γ, Q), has to be used to pay for the work of the evaluation of e_1 and the work of the evaluation of e_2. Let us consider a concrete example again.

```
mult2(ys) = let xs = mult(496,ys) in
            let zs = mult(8128,ys) in (xs,zs)
```

The work (and depth) of the evaluation of the expression $\mathsf{mult2}(\mathsf{ys})$ is $2|\mathsf{ys}|$ in the metric M^*. In the type judgement, we express this bound as follows. First, we type the two function applications of mult as before using

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \vdash^{M^*} \mathsf{mult}(\mathsf{x},\mathsf{ys}) : (L(\mathrm{int}), Q')$$

where $p_Q(n,a) = |a|$ and $p_{Q'}(a) = 0$. In the type judgement

$$\mathsf{ys}{:}L(\mathrm{int}); R \vdash^{M^*} \mathsf{mult2}(\mathsf{ys}) : (L(\mathrm{int}) * L(\mathrm{int}), R')$$

we require that $p_R(a) \geqslant p_Q(a) + p_Q(a)$, that is, the initial potential (defined by the coefficients R) has to be shared in the two sequential branches. Such a sharing can still be expressed with linear constraints. such as $r_{[*]} \geqslant q_{(*,[*])} + q_{(*,[*])}$. A valid instantiation of R would thus correspond to the potential function $p_R(a) = 2|a|$. With this instantiation, the previous typing reflects the bound $2|\mathsf{ys}|$ for the evaluation of $\mathsf{mult2}(\mathsf{ys})$.

A slightly more involved example is the function $\mathsf{dyad} : L(\mathrm{int}) * L(\mathrm{int}) \to L(L(\mathrm{int}))$ which computes the dyadic product of two integer lists.

```
dyad (u,v) = match u with | nil  → nil
     | (x::xs)  → let x' = mult(x,v) in
                  let xs' = dyad(xs,v) in x'::xs';
```

Using the metric M^* that counts multiplications, multivariate resource analysis for sequential programs derives the bound $|\mathsf{u}|\cdot|\mathsf{v}|$. In the cons branch of the pattern match, we have the potential $|\mathsf{xs}|\cdot|\mathsf{v}| + |\mathsf{v}|$ which is shared to pay for the cost $|\mathsf{v}|$ of $\mathsf{mult}(\mathsf{x},\mathsf{v})$ and the cost $|\mathsf{xs}|\cdot|\mathsf{v}|$ of $\mathsf{dyad}(\mathsf{xs},\mathsf{v})$.

Moving multivariate potential through a program is not trivial; especially in the presence of nested data structures like trees of lists. To give an idea of the challenges, consider the expression e that is defined as follows.

```
let xs = mult(496,ys) in
let zs = append(ys,ys) in dyad(xs,zs)
```

The depth of evaluating e in the metric M^* is bounded by $|\mathsf{ys}| + 2|\mathsf{ys}|^2$. Like in the previous example, we express this in amortized resource analysis with the initial potential $|\mathsf{ys}| + 2|\mathsf{ys}|^2$. This potential has to be shared to pay for the cost of the evaluations of $\mathsf{mult}(496,\mathsf{ys})$ (namely $|\mathsf{ys}|$) and $\mathsf{dyad}(\mathsf{xs},\mathsf{zs})$ (namely $2|\mathsf{ys}|^2$). However, the type of dyad requires the quadratic potential $|\mathsf{xs}|\cdot|\mathsf{zs}|$. In this simple example, it is easy to see that $|\mathsf{xs}|\cdot|\mathsf{zs}| = 2|\mathsf{ys}|^2$. But in general, it is not straightforward to compute such a conversion of potential in an automatic analysis system, especially for nested data structures and super-linear size changes. The type inference for multivariate amortized resource analysis for sequential programs can analyze such programs efficiently [7].

Parallel Composition. The insight of this paper is that the potential method works also well to derive bounds on parallel evaluations. The main challenge in the development of an amortized resource analysis for parallel evaluations is to ensure the same compositionality as in sequential amortized resource analysis.

The basic idea of our new analysis system is to allow each branch in a parallel evaluation to use all the available potential without sharing. Consider for example the previously defined function mult2 in which we evaluate the two applications of mult in parallel.

```
mult2par(ys) = par xs = mult(496,ys)
               and zs = mult(8128,ys) in (xs,zs)
```

Since the depth of $\mathsf{mult}(\mathsf{n},\mathsf{ys})$ is $|\mathsf{ys}|$ for every n and the two applications of mult are evaluated in parallel, the depth of the evaluation of $\mathsf{mult2par}(\mathsf{ys})$ is $|\mathsf{ys}|$ in the metric M^*.

In the type judgement, we type the two function applications of mult as in the sequential case in which

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \mid\!\!\!\overline{-}^{M^*} \mathsf{mult}(\mathsf{x},\mathsf{ys}) : (L(\mathrm{int}), Q')$$

such that $p_Q(n,a) = |a|$ and $p_{Q'}(a) = 0$. In the type judgement

$$\mathsf{ys}{:}L(\mathrm{int}); R \mid\!\!\!\overline{-}^{M^*} \mathsf{mult2par}(\mathsf{ys}) : (L(\mathrm{int}) * L(\mathrm{int}), R')$$

for mult2par we require however only that $p_R(a) \geqslant p_Q(a)$. In this way, we express that the initial potential defined by the coefficients R has to be sufficient to

cover the cost of each parallel branch. Consequently, a possible instantiation of R corresponds to the potential function $p_R(a) = |a|$.

In the function dyad, we can replace the sequential computation of the inner lists of the result by a parallel computation in which we perform all calls to the function mult in parallel. The resulting function is dyad_par.

```
dyad_par (u,v) = match u with | nil  → nil
                 | (x::xs)  → par x' = mult(x,v)
                              and xs' = dyad_par(xs,v) in x'::xs';
```

The depth of dyad_par is |v|. In the type-based amortized analysis, we hence start with the initial potential |v|. In the cons branch of the pattern match, we can use the initial potential to pay for both, the cost |v| of mult(x, v) and the cost |v| of the recursive call dyad(xs, v) without sharing the initial potential.

Unfortunately, the compositionality of the sequential system is not preserved by this simple idea. The problem is that the naive reuse of potential that is passed through parallel branches would break the soundness of the system. To see why, consider the following function.

```
mult4(ys) = par xs = mult(496,ys)
  and zs = mult(8128,ys) in (mult(5,xs), mult(10,zs))
```

Recall, that a valid typing for xs = mult(496, ys) could take the initial potential 2|ys| and assign the potential |xs| to the result. If we would simply reuse the potential 2|ys| to type the second application of mult in the same way then we would have the potential |xs| + |zs| after the parallel branches. This potential could then be used to pay for the cost of the remaining two applications of mult. We have now verified the unsound bound 2|ys| on the depth of the evaluation of the expression mult4(ys) but the depth of the evaluation is 3|ys|.

The problem in the previous reasoning is that we doubled the part of the initial potential that we passed on for later use in the two parallel branches of the parallel composition. To fix this problem, we need a separate analysis of the sizes of data structures and the cost of parallel evaluations.

In this paper, we propose to use cost-free type judgements to reason about the size changes in parallel branches. Instead of simply using the initial potential in both parallel branches, we share the potential between the two branches but analyze the two branches twice. In the first analysis, we only pay for the resource consumption of the first branch. In the second, analysis we only pay for resource consumption of the second branch.

A cost-free type judgement is like any other type judgement in amortized resource analysis but uses the cost-free metric cf that assigns zero cost to every evaluation step. For example, a cost-free typing of the function mult(ys) would express that the initial potential can be passed to the result of the function. In the cost-free typing judgement

$$\mathsf{x}{:}\mathrm{int}, \mathsf{ys}{:}L(\mathrm{int}); Q \vdash^{\mathsf{cf}} \mathsf{mult}(\mathsf{x},\mathsf{ys}) : (L(\mathrm{int}), Q')$$

a valid instantiation of Q and Q' would correspond to the potential

$$p_Q(n,a) = |a| \qquad \text{and} \qquad p_{Q'}(a) = |a| \ .$$

The intuitive meaning is that in a call zs = mult(x, ys), the initial potential $|ys|$ can be transformed to the potential $|zs|$ of the result.

Using cost-free typings, we can now correctly reason about the depth of the evaluation of mult4. We start with the initial potential $3|\mathsf{ys}|$ and have to consider two cases in the parallel binding. In the first case, we have to pay only for resource cost of $\mathsf{mult}(\mathsf{496}, \mathsf{ys})$. So we share the initial potential and use $2|\mathsf{ys}|$: $|\mathsf{ys}|$ to pay the cost of $\mathsf{mult}(\mathsf{496}, \mathsf{ys})$ and $|\mathsf{ys}|$ to assign the potential $|\mathsf{xs}|$ to the result of the application. The reminder $|\mathsf{ys}|$ of the initial potential is used in a cost-free typing of $\mathsf{mult}(\mathsf{8128}, \mathsf{ys})$ where we assign the potential $|\mathsf{zs}|$ to the result of the function without paying any evaluation cost. In the second case, we derive a similar typing in which the roles of the two function calls are switched. In both cases, we start with the potential $3|\mathsf{ys}|$ and end with the potential $|\mathsf{xs}| + |\mathsf{zs}|$. We use it to pay for the two remaining calls of mult and have verified the correct bound.

In the univariate case, using the notation from [3, 19], we could formulate the type rule for parallel composition as follows. Here, the coefficients Q are not globally attached to a type or context but appear locally at list types such as $L^q(\text{int})$. The sharing operator $\Gamma \curlyvee (\Gamma_1, \Gamma_2, \Gamma_3)$ requires the sharing of the potential in the context Γ in the contexts Γ_1,Γ_2 and Γ_3. For instance, we have $x{:}L^6(\text{int}) \curlyvee (x{:}L^2(\text{int}), x{:}L^3(\text{int}), x{:}L^1(\text{int}))$.

$$\frac{\begin{array}{c}\Gamma \curlyvee (\Delta_1, \Gamma_2, \Gamma') \qquad \Gamma \curlyvee (\Gamma_1, \Delta_2, \Gamma') \qquad \Gamma_1 \vdash^{M} e_1 : A_1 \qquad \Delta_2 \vdash^{\mathsf{cf}} e_2 : A_2 \\ \Delta_1 \vdash^{\mathsf{cf}} e_1 : A_1 \qquad \Gamma_2 \vdash^{M} e_2 : A_2 \qquad \Gamma', x_1{:}A_1, x_2{:}A_2 \vdash^{M} e : B\end{array}}{\Gamma \vdash^{M} \mathsf{par}\, x_1 = e_1 \,\mathsf{and}\, x_2 = e_2 \,\mathsf{in}\, e : B}$$

In the rule, the initial potential Γ is shared twice using the sharing operator $\curlyvee$. First, to pay the cost of evaluating e_2 and e, and to pass potential to x_1 using the cost-free type judgement $\Delta_1 \vdash^{\mathsf{cf}} e_1 : A_1$. Second, to pay the cost of evaluation e_1 and e, and to pass potential to x_2 via the judgement $\Delta_2 \vdash^{\mathsf{cf}} e_2 : A_2$.

This work generalizes the idea to multivariate resource polynomials for which we also have to deal with mixed potential such as $|x_1|\cdot|x_2|$. The approach features the same compositionality as the sequential version of the analysis. As the experiments in Section 7 show, the analysis works well for many typical examples.

The use of cost-free typings to separate the reasoning about size changes of data structures and resource cost in amortized analysis has applications that go beyond parallel evaluations. Similar problems arise in sequential (and parallel) programs when deriving bounds for non-additive cost such as stack-space usage or recursion depth. We envision that the developed technique can be used to derive bounds for these cost measures too.

Other Forms of Parallelism. The binary parallel binding is a simple yet powerful form of parallelism. However, it is (for example) not possible to directly implement NESL's model of sequences that allows to perform an operation for every element in the sequence in constant depth. The reason is that the parallel binding would introduce a linear overhead.

Nevertheless it is possible to introduce another binary parallel binding that is semantically equivalent except that it has zero depth cost. We can then analyze more powerful parallelism primitives by translating them into code that uses this cost-free parallel binding. To demonstrate such a translation, we implemented NESL's [15] parallel sequence comprehensions in RAML (see Section 6).

4 Resource Polynomials and Annotated Types

In this section, we introduce multivariate resource polynomials and annotated types. Our goal is to systematically describe the potential functions that map data structures to non-negative rational numbers. Multivariate resource polynomials are a generalization of non-negative linear combinations of binomial coefficients. They have properties that make them ideal for the generation of succinct linear constraint systems in an automatic amortized analysis. The presentation might appear quite low level but this level of detail is necessary to describe the linear constraints in the type rules.

Two main advantages of resource polynomials are that they can express more precise bounds than non-negative linear-combinations of standard polynomials and that they can succinctly describe common size changes of data that appear in construction and destruction of data. More explanations can be found in the previous literature on multivariate amortized resource analysis [13, 7].

4.1 Resource Polynomials

A resource polynomial maps a value of some data type to a nonnegative rational number. Potential functions and thus resource bounds are always resource polynomials.

Base Polynomials. For each data type A we first define a set $P(A)$ of functions $p : [\![A]\!] \to \mathbb{N}$ that map values of type A to natural numbers. These *base polynomials* form a basis (in the sense of linear algebra) of the resource polynomials for type A. The resource polynomials for type A are then given as nonnegative rational linear combinations of the base polynomials. We define $P(A)$ as follows.

$$P(\text{int}) = \{a \mapsto 1\} \qquad P(A_1 * A_2) = \{(a_1, a_2) \mapsto p_1(a_1) \cdot p_2(a_2) \mid p_i \in P(A_i)\}$$
$$P(L(A)) = \{\Sigma\Pi[p_1, \ldots, p_k] \mid k \in \mathbb{N}, p_i \in P(A)\}$$

We have $\Sigma\Pi[p_1, \ldots, p_k]([a_1, \ldots, a_n]) = \sum_{1 \leqslant j_1 < \cdots < j_k \leqslant n} \prod_{1 \leqslant i \leqslant k} p_i(a_{j_i})$. Every set $P(A)$ contains the constant function $v \mapsto 1$. For lists $L(A)$ this arises for $k = 0$ (one element sum, empty product).

For example, the function $\ell \mapsto \binom{|\ell|}{k}$ is in $P(L(A))$ for every $k \in \mathbb{N}$; simply take $p_1 = \ldots = p_k = 1$ in the definition of $P(L(A))$. The function $(\ell_1, \ell_2) \mapsto \binom{|\ell_1|}{k_1} \cdot \binom{|\ell_2|}{k_2}$ is in $P(L(A) * L(B))$ for every $k_1, k_2 \in \mathbb{N}$ and $[\ell_1, \ldots, \ell_n] \mapsto \sum_{1 \leqslant i < j \leqslant n} \binom{|\ell_i|}{k_1} \cdot \binom{|\ell_j|}{k_2} \in P(L(L(A)))$ for every $k_1, k_2 \in \mathbb{N}$.

Resource Polynomials. A *resource polynomial* $p : [\![A]\!] \to \mathbb{Q}_0^+$ for a data type A is a non-negative linear combination of base polynomials, i.e., $p = \sum_{i=1,\ldots,m} q_i \cdot p_i$ for $q_i \in \mathbb{Q}_0^+$ and $p_i \in P(A)$. $R(A)$ is the set of resource polynomials for A.

An instructive, but not exhaustive, example is given by $R_n = R(L(\text{int}) * \cdots * L(\text{int}))$. The set R_n is the set of linear combinations of products of binomial coefficients over variables $x_1, \ldots, x_n$, that is, $R_n = \{\sum_{i=1}^{m} q_i \prod_{j=1}^{n} \binom{x_j}{k_{ij}} \mid q_i \in \mathbb{Q}_0^+, m \in \mathbb{N}, k_{ij} \in \mathbb{N}\}$. Concrete examples that illustrate the definitions follow in the next subsection.

4.2 Annotated Types

To relate type annotations in the type system to resource polynomials, we introduce names (or indices) for base polynomials. These names are also helpful to intuitively explain the base polynomials of a given type.

Names For Base Polynomials. To assign a unique name to each base polynomial we define the *index set* $\mathcal{I}(A)$ to denote resource polynomials for a given data type A. Essentially, $\mathcal{I}(A)$ is the meaning of A with every atomic type replaced by the *unit index* $\circ$.

$$\mathcal{I}(\text{int}) = \{\circ\} \qquad \mathcal{I}(A_1 * A_2) = \{(i_1, i_2) \mid i_1 \in \mathcal{I}(A_1) \text{ and } i_2 \in \mathcal{I}(A_2)\}$$
$$\mathcal{I}(L(A)) = \{[i_1, \dots, i_k] \mid k \geqslant 0, i_j \in \mathcal{I}(A)\}$$

The *degree* $\deg(i)$ of an index $i \in \mathcal{I}(A)$ is defined as follows.

$$\deg(\circ) = 0 \qquad \deg(i_1, i_2) = \deg(i_1) + \deg(i_2)$$
$$\deg([i_1, \dots, i_k]) = k + \deg(i_1) + \dots + \deg(i_k)$$

Let $\mathcal{I}_k(A) = \{i \in \mathcal{I}(A) \mid \deg(i) \leqslant k\}$. The indices $i \in \mathcal{I}_k(A)$ are an enumeration of the base polyonomials $p_i \in P(A)$ of degree at most k. For each $i \in \mathcal{I}(A)$, we define a base polynomial $p_i \in P(A)$ as follows: If $A = \text{int}$ then $p_\circ(v) = 1$. If $A = (A_1 * A_2)$ is a pair type and $v = (v_1, v_2)$ then $p_{(i_1,i_2)}(v) = p_{i_1}(v_1) \cdot p_{i_2}(v_2)$. If $A = L(B)$ is a list type and $v \in [\![L(B)]\!]$ then $p_{[i_1,\dots,i_m]}(v) = \Sigma\Pi[p_{i_1}, \dots, p_{i_m}](v)$. We use the notation 0_A (or just 0) for the index in $\mathcal{I}(A)$ such that $p_{0_A}(a) = 1$ for all a. We have $0_{\text{int}} = \circ$ and $0_{(A_1 * A_2)} = (0_{A_1}, 0_{A_2})$ and $0_{L(B)} = []$. If $A = L(B)$ for a data type B then the index $[0, \dots, 0] \in \mathcal{I}(A)$ of length n is denoted by just n. We identify the index (i_1, i_2, i_3, i_4) with the index $(i_1, (i_2, (i_3, i_4)))$.

Examples. First consider the type int. The index set $\mathcal{I}(\text{int}) = \{\circ\}$ only contains the unit element because the only base polynomial for the type int is the constant polynomial $p_\circ : \mathbb{Z} \to \mathbb{N}$ that maps every integer to 1, that is, $p_\circ(n) = 1$ for all $n \in \mathbb{Z}$. In terms of resource-cost analysis this implies that the resource polynomials can not represent cost that depends on the value of an integer.

Now consider the type $L(\text{int})$. The index set for lists of integers is $\mathcal{I}(L(\text{int})) = \{[], [\circ], [\circ, \circ], \dots\}$, the set of lists of unit indices $\circ$. The base polynomial $p_{[]} : [\![L(\text{int})]\!] \to \mathbb{N}$ is defined as $p_{[]}([a_1, \dots, a_n]) = 1$ (one element sum and empty product). More interestingly, we have $p_{[\circ]}([a_1, \dots, a_n]) = \sum_{1 \leqslant j \leqslant n} 1 = n$ and $p_{[\circ,\circ]}([a_1, \dots, a_n]) = \sum_{1 \leqslant j_1 < j_2 \leqslant n} 1 = \binom{n}{2}$. In general, if $i_k = [\circ, \dots, \circ]$ is as list with k unit indices then $p_{i_k}([a_1, \dots, a_n]) = \sum_{1 \leqslant j_1 < \dots < j_k \leqslant n} 1 = \binom{n}{k}$. The intuition is that the base polynomial $p_{i_k}([a_1, \dots, a_n])$ describes a constant resource cost that arises for every ordered k-tuple $(a_{j_1}, \dots, a_{j_n})$.

Finally, consider the type $L(L(\text{int}))$ of lists of lists of integers. The corresponding index set is $\mathcal{I}(L(L(\text{int}))) = \{[]\} \cup \{[i] \mid i \in \mathcal{I}(L(\text{int}))\} \cup \{[i_1, i_2] \mid i_1, i_2 \in \mathcal{I}(L(\text{int}))\} \cup \dots$. Again we have $p_{[]} : [\![L(L(\text{int}))]\!] \to \mathbb{N}$ and $p_{[]}([a_1, \dots, a_n]) = 1$. Moreover we also get the binomial coefficients again: If the index $i_k = [[], \dots, []]$ is as list of k empty lists then $p_{i_k}([a_1, \dots, a_n]) = \sum_{1 \leqslant j_1 < \dots < j_k \leqslant n} 1 = \binom{n}{k}$. This describes a cost that would arise in a program that computes something of constant cost for tuples of inner lists (e.g., sorting with respect to the smallest head elements). However, the base polynomials can also refer to the lengths of the inner

lists. For instance, we have $p[[\circ,\circ]]([a_1,\ldots,a_n]) = \sum_{1\leqslant i\leqslant n}\binom{|a_i|}{2}$, which represents a quadratic cost for every inner list (e.g, sorting the inner lists). This is not to be confused with the base polynomial $p_{[\circ,\circ]}([a_1,\ldots,a_n]) = \sum_{1\leqslant i<j\leqslant n}|a_i||a_j|$, which can be used to account for the cost of the comparisons in a lexicographic sorting of the outer list.

Annotated Types and Potential Functions. We use the indices and base polynomials to define type annotations and resource polynomials. We then give examples to illustrate the definitions.

A *type annotation* for a data type A is defined to be a family

$$Q_A = (q_i)_{i\in\mathcal{I}(A)} \text{ with } q_i \in \mathbb{Q}_0^+$$

We say Q_A is of *degree (at most) k* if $q_i = 0$ for every $i \in \mathcal{I}(A)$ with $\deg(i) > k$. An *annotated data type* is a pair (A, Q_A) of a data type A and a type annotation Q_A of some degree k.

Let H be a heap and let ℓ be a location with $H \vDash \ell{\mapsto}a : A$ for a data type A. Then the type annotation Q_A defines the *potential* $\Phi_H(\ell{:}(A, Q_A)) = \sum_{i\in\mathcal{I}(A)} q_i \cdot p_i(a)$. If $a \in [\![A]\!]$ and Q is a type annotation for A then we also write $\Phi(a : (A, Q))$ for $\sum_i q_i p_i(a)$.

Let for example, $Q = (q_i)_{i\in L(\mathrm{int})}$ be an annotation for the type $L(\mathrm{int})$ and let $q_{[]} = 2$, $q_{[\circ]} = 2.5$, $q_{[\circ,\circ,\circ]} = 8$, and $q_i = 0$ for all other $i \in \mathcal{I}(L(\mathrm{int}))$. The we have $\Phi([a_1,\ldots,a_n] : (L(\mathrm{int}), Q)) = 2 + 2.5n + 8\binom{n}{3}$.

The Potential of a Context. For use in the type system we need to extend the definition of resource polynomials to typing contexts. We treat a context like a tuple type. Let $\Gamma = x_1{:}A_1,\ldots,x_n{:}A_n$ be a typing context and let $k \in \mathbb{N}$. The index set $\mathcal{I}(\Gamma)$ is defined through $\mathcal{I}(\Gamma) = \{(i_1,\ldots,i_n) \mid i_j \in \mathcal{I}(A_j)\}$.

The degree of $i = (i_1,\ldots,i_n) \in \mathcal{I}(\Gamma)$ is defined through $\deg(i) = \deg(i_1) + \cdots + \deg(i_n)$. As for data types, we define $I_k(\Gamma) = \{i \in I(\Gamma) \mid \deg(i) \leqslant k\}$. A *type annotation* Q for Γ is a family $Q = (q_i)_{i\in\mathcal{I}_k(\Gamma)}$ with $q_i \in \mathbb{Q}_0^+$. We denote a *resource-annotated context* with $\Gamma; Q$. Let H be a heap and V be a stack with $H \vDash V : \Gamma$ where $H \vDash V(x_j){\mapsto}a_{x_j} : \Gamma(x_j)$.

The potential of an annotated context $\Gamma; Q$ with respect to then environment H and V is $\Phi_{V,H}(\Gamma; Q) = \sum_{(i_1,\ldots,i_n)\in\mathcal{I}_k(\Gamma)} q_{\vec{i}} \prod_{j=1}^n p_{i_j}(a_{x_j})$. In particular, if $\Gamma = \varnothing$ then $\mathcal{I}_k(\Gamma) = \{()\}$ and $\Phi_{V,H}(\Gamma; q_{()}) = q_{()}$. We sometimes also write q_0 for $q_{()}$.

5 Type System for Bounds on the Depth

In this section, we formally describe the novel resource-aware type system. We focus on the type judgement and explain the rules that are most important for handling parallel evaluation. The full type system is given in the extended version of this article [17].

The main theorem of this section proves the soundness of the type system with respect to the depths of evaluations as defined by the operational big-step semantics. The soundness holds for terminating and non-terminating evaluations.

Type Judgments. The typing rules in Figure 2 define a *resource-annotated typing judgment* of the form

$$\Sigma; \Gamma; \{Q_1,\ldots,Q_n\} \overset{M}{\vdash} e : (A, Q')$$

where M is a metric, $n \in \{1, 2\}$, e is an expression, Σ is a resource-annotated signature (see below), $(\Gamma; Q_i)$ is a resource-annotated context for every $i \in \{1, \ldots, n\}$, and (A, Q') is a resource-annotated data type. The intended meaning of this judgment is the following. If there are more than $\Phi(\Gamma; Q_i)$ resource units available for every $i \in \{1, \ldots, n\}$ then this is sufficient to pay for the depth of the evaluation of e under the metric M. In addition, there are more than $\Phi(v{:}(A, Q'))$ resource units left if e evaluates to a value v.

In outermost judgements, we are only interested in the case where $n = 1$ and the judgement is equivalent to the similar judgement for sequential programs [7]. The form in which $n = 2$ is introduced in the type rule E:Par for parallel bindings and eliminated by multiple applications of the sharing rule E:Share (more explanations follow).

The type judgement is affine in the sense that every variable in a context Γ can be used at most once in the expression e. Of course, we have to also deal with expressions in which a variable occurs more than once. To account for multiple variable uses we use the sharing rule T:Share that doubles a variable in a context without increasing the potential of the context.

As usual Γ_1, Γ_2 denotes the union of the contexts Γ_1 and Γ_2 provided that $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$. We thus have the implicit side condition $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \varnothing$ whenever Γ_1, Γ_2 occurs in a typing rule. Especially, writing $\Gamma = x_1{:}A_1, \ldots, x_k{:}A_k$ means that the variables x_i are pairwise distinct.

Programs with Annotated Types. *Resource-annotated first-order types* have the form $(A, Q) \to (B, Q')$ for annotated data types (A, Q) and (B, Q'). A *resource-annotated signature* Σ is a finite, partial mapping of function identifiers to *sets of* resource-annotated first-order types. A program with resource-annotated types for the metric M consists of a resource-annotated signature Σ and a family of expressions with variables identifiers $(e_f, y_f)_{f \in \mathrm{dom}(\Sigma)}$ such that $\Sigma; y_f{:}A; Q \overset{M}{\vdash} e_f : (B, Q')$ for every function type $(A, Q) \to (B, Q') \in \Sigma(f)$.

Sharing. Let $\Gamma, x_1{:}A, x_2{:}A; Q$ be an annotated context. The *sharing operation* $\curlyvee Q$ defines an annotation for a context of the form $\Gamma, x{:}A$. It is used when the potential is split between multiple occurrences of a variable. Details can be found in the full version of the article.

Typing Rules. Figure 2 shows the annotated typing rules that are most relevant for parallel evaluation. Most of the other rules are similar to the rules for multivariate amortized analysis for sequential programs [13, 20]. The main difference it that the rules here operate on annotations that are singleton sets $\{Q\}$ instead of the usual context annotations Q.

In the rules T:Let and T:Par, the result of the evaluation of an expression e is bound to a variable x. The problem that arises is that the resulting annotated context $\Delta, x{:}A, Q'$ features potential functions whose domain consists of data that is referenced by x as well as data that is referenced by Δ. This potential has to be related to data that is referenced by Δ and the free variables in e.

To express the relations between mixed potentials before and after the evaluation of e, we introduce a new auxiliary binding judgement of the from

$$\Sigma; \Gamma, \Delta; Q \overset{M}{\vdash} e \rightsquigarrow \Delta, x{:}A; Q'$$

$$\frac{\Sigma;\Gamma_1,\Gamma_2;R \mid\!\!\!\frac{M}{}\ e_1 \rightsquigarrow \Gamma_2, x{:}A; R' \qquad \Sigma;,\Gamma_2,x{:}A;\{R'\} \mid\!\!\!\frac{M}{}\ e_2 : (B,Q') \qquad Q = R + M^{\mathsf{let}}}{\Sigma;\Gamma_1,\Gamma_2;\{Q\} \mid\!\!\!\frac{M}{}\ \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : (B,Q')}\ (\text{T:Let})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma_1,\Gamma_2,\Delta;P \mid\!\!\!\frac{\mathsf{cf}}{}\ e_1 \rightsquigarrow \Gamma_2,\Delta,x_1{:}A_1;P' \\ \Sigma;\Gamma_2,\Delta,x_1{:}A_1;P' \mid\!\!\!\frac{M}{}\ e_2 \rightsquigarrow \Delta,x_1{:}A_1,x_2{:}A_2;R \\ \Sigma;\Gamma_2,\Delta,x_1{:}A_1;Q' \mid\!\!\!\frac{\mathsf{cf}}{}\ e_2 \rightsquigarrow \Delta,x_1{:}A_1,x_2{:}A_2;R \\ \Sigma;\Gamma_1,\Gamma_2,\Delta;Q \mid\!\!\!\frac{M}{} e_1 \rightsquigarrow \Gamma_2,\Delta,x_1{:}A_1;Q' \qquad \Sigma;\Delta,x_1{:}A_1,x_2{:}A_2;R \mid\!\!\!\frac{M}{} e : (B,R')\end{array}}{\Sigma;\Gamma_1,\Gamma_2,\Delta;\{Q+M^{\mathsf{Par}},P+M^{\mathsf{Par}}\} \mid\!\!\!\frac{M}{}\ \mathsf{par}\, x_1 = e_1\, \mathsf{and}\, x_2 = e_2\, \mathsf{in}\, e : (B,R')}\ (\text{T:Par})$$

$$\frac{\Sigma;\Gamma,x_1{:}A,x_2{:}A;\{P_1,\ldots,P_m\} \mid\!\!\!\frac{M}{}\ e : (B,Q') \qquad \forall i\, \exists j : Q_j = \curlyvee P_i}{\Sigma;\Gamma,x{:}A;\{Q_1,\ldots,Q_n\} \mid\!\!\!\frac{M}{}\ e[x/x_1,x/x_2] : (B,Q')}\ (\text{T:Share})$$

◆ ◆ ◆

$$\frac{\begin{array}{c}\forall j \in \mathcal{I}(\Delta):\quad j{=}\vec{0} \implies \Sigma;\Gamma;\pi_j^{\Gamma}(Q) \mid\!\!\!\frac{M}{}\ e : (A,\pi_j^{x:A}(Q')) \\ j{\neq}\vec{0} \implies \Sigma_j;\Gamma;\pi_j^{\Gamma}(Q) \mid\!\!\!\frac{\mathsf{cf}}{}\ e : (A,\pi_j^{x:A}(Q'))\end{array}}{\Sigma;\Gamma,\Delta;Q \mid\!\!\!\frac{M}{}\ e \rightsquigarrow \Delta,x{:}A;Q'}\ (\text{B:Bind})$$

Fig. 2. Selected novel typing rules for annotated types and the binding rule for multivariate variable binding.

in the rule B:Bind. The intuitive meaning of the judgement is the following. Assume that e is evaluated in the context Γ,Δ, that $\mathrm{FV}(e) \in \mathrm{dom}(\Gamma)$, and that e evaluates to a value that is bound to the variable x. Then the initial potential $\Phi(\Gamma,\Delta;Q)$ is larger than the cost of evaluating e in the metric M plus the potential of the resulting context $\Phi(\Delta,x{:}A;Q')$.

The rule T:Par for parallel bindings $\mathsf{par}\, x_1 = e_1\, \mathsf{and}\, x_2 = e_2\, \mathsf{in}\, e$ is the main novelty in the type system. The idea is that we type the expressions e_1 and e_2 twice using the new binding judgement. In the first group of bindings, we account for the cost of e_1 and derive a context $\Gamma_2,\Delta,x_1{:}A_1;P_1'$ in which the result of the evaluation of e_1 is bound to x_1. This context is then used to bind the result of evaluating e_2 in the context $\Delta,x_1{:}A_1,x_2{:}A_2;R$ without paying for the resource consumption. In the second group of bindings, we also derive the context $\Delta,x_1{:}A_1,x_2{:}A_2;R$ but pay for the cost of evaluating e_2 instead of e_1. The type annotations Q_1 and Q_2 for the initial context $\Gamma = \Gamma_1,\Gamma_2,\Delta$ establish a bound on the depth d of evaluating the whole parallel binding: If the depth of evaluating e_1 is larger than the depth of evaluating e_2 then $\Phi(\Gamma;Q_1) \geqslant d$. Otherwise we have $\Phi(\Gamma;Q_2) \geqslant d$. If the parallel binding evaluates to a value v then we have additionally that $\max(\Phi(\Gamma;Q_1),\Phi(\Gamma;Q_2)) \geqslant d + \Phi(v{:}(B,Q'))$.

It is important that the annotations Q_1 and Q_2 of the initial context Γ_1,Γ_2,Δ can defer. The reason is that we have to allow a different sharing of potential in the two groups of bindings. If we would require $Q_1 = Q_2$ then the system would be too restrictive. However, each type derivation has to establish the equality of the two annotations directly after the use of T:Par by multiple uses of the

sharing rule T:Share. Note that T:Par is the only rule that can introduce a non-singleton set $\{Q_1, Q_n\}$ of context annotations.

T:Share has to be applied to expressions that contain a variable twice (x in the rule). The sharing operation $\curlyvee P$ transfers the annotation P for the context $\Gamma, x_1{:}A, x_2{:}A$ into an annotation Q for the context $\Gamma, x{:}A$ without loss of potential . This is crucial for the accuracy of the analysis since instances of T:Share are quite frequent in typical examples. The remaining rules are affine in the sense that they assume that every variable occurs at most once in the typed expression.

T:Share is the only rule whose premiss allows judgements that contain a non-singleton set $\{P_1, \dots, P_m\}$ of context annotations. It has to be applied to produce a judgement with singleton set $\{Q\}$ before any of the other rules can be applied. The idea is that we always have $n \leqslant m$ for the set $\{Q_1, \dots, Q_n\}$ and the sharing operation $\curlyvee_i$ is used to unify the different P_i.

Soundness. The operational big-step semantics with partial evaluations makes it possible to state and prove a strong soundness result. An annotated type judgment for an expression e establishes a bound on the depth of all evaluations of e in a well-formed environment; regardless of whether these evaluations diverge or fail.Moreover, the soundness theorem states also a stronger property for terminating evaluations. If an expression e evaluates to a value v in a well-formed environment then the difference between initial and final potential is an upper bound on the depth of the evaluation.

Theorem 3 (Soundness). *If $H \vDash V{:}\Gamma$ and $\Sigma; \Gamma; \mathcal{Q} \vdash e{:}(B, Q')$ then there exists a $Q \in \mathcal{Q}$ such that the following holds.*

1. *If $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ then $d \leqslant \Phi_{V,H}(\Gamma; Q) - \Phi_{H'}(\ell{:}(B, Q'))$.*
2. *If $V, H \overset{M}{\vdash} e \Downarrow \rho \mid (w, d)$ then $d \leqslant \Phi_{V,H}(\Gamma; Q)$.*

Theorem 3 is proved by a nested induction on the derivation of the evaluation judgment and the type judgment $\Gamma; Q \vdash e{:}(B, Q')$. The inner induction on the type judgment is needed because of the structural rules. There is one proof for all possible instantiations of the resource constants.

The proof of most rules is very similar to the proof of the rules for multivariate resource analysis for sequential programs [7]. The main novelty is the treatment of parallel evaluation in the rule T:Par which we described previously.

If the metric M is simple (all constants are 1) then it follows from Theorem 3 that the bounds on the depth also prove the termination of programs.

Corollary 1. *Let M be a simple metric. If $H \vDash V{:}\Gamma$ and $\Sigma; \Gamma; Q \vdash e{:}(A, Q')$ then there are $w \in \mathbb{N}$ and $d \leqslant \Phi_{V,H}(\Gamma; Q)$ such that $V, H \overset{M}{\vdash} e \Downarrow (\ell, H') \mid (w, d)$ for some ℓ and H'.*

Type Inference. In principle, type inference consists of four steps. First, we perform a classic type inference for the simple types such as nat array. Second, we fix a maximal degree of the bounds and annotate all types in the derivation of the simple types with variables that correspond to type annotations for resource polynomials of that degree. Third, we generate a set of linear inequalities, which express the relationships between the added annotation variables as specified by

the type rules. Forth, we solve the inequalities with an LP solver such as CLP. A solution of the linear program corresponds to a type derivation in which the variables in the type annotations are instantiated according to the solution.

In practice, the type inference is slightly more complex. Most importantly, we have to deal with resource-polymorphic recursion in many examples. This means that we need a type annotation in the recursive call that differs from the annotation in the argument and result types of the function. To infer such types we successively infer type annotations of higher and higher degree. Details can be found in previous work [21]. Moreover, we have to use algorithmic versions of the type rules in the inference in which the non-syntax-directed rules are integrated into the syntax-directed ones [7]. Finally, we use several optimizations to reduce the number of generated constraints. See [7] for an example type derivation.

6 Nested Data Parallelism

The techniques that we describe in this work for a minimal function language scale to more advanced parallel languages such as Blelloch's NESL [15].

To describe the novel type analysis in this paper, we use a binary binding construct to introduce parallelism. In NESL, parallelism is introduced via built-in functions on sequences as well as parallel sequence comprehension that is similar to Haskell's list comprehension. The depth of all built-in sequence functions such as *append* and *sum* is constant in NESL. Similarly, the depth overhead of the parallel sequence comprehension is constant too. Of course, it is possible to define equivalent functions in RAML. However, the depth would often be linear since we, for instance, have to sequentially form the resulting list.

Nevertheless, the user definable resource metrics in RAML make it easy to introduce built-in functions and language constructs with customized work and depth. For instance we could implement NESL's *append* like the recursive append in RAML but use a metric inside the function body in which all evaluation steps have depth zero. Then the depth of the evaluation of $\mathsf{append}(\mathsf{x},\mathsf{y})$ is constant and the work is linear in $|x|$.

To demonstrate this ability of our approach, we implemented parallel list comprehensions, NESL's most powerful construct for parallel computations. A list comprehension has the form $\{\ e\ :\ x_1 \,\mathsf{in}\, e_1\,;\ldots;\, x_n \,\mathsf{in}\, e_n\ \mid\ e_b\ \}$. where e is an expression, $e_1,\ldots,e_n$ are expressions of some list type, and e_b is a boolean expression. The semantics is that we bind $x_1,\ldots,x_n$ successively to the elements of the lists $e_1,\ldots,e_n$ and evaluate e_b and e under these bindings. If e_b evaluates to true under a binding then we include the result of e under that binding in the resulting list. In other words, the above list comprehension is equivalent to the Haskell expression $[\ e \mid (x_1,\ldots,x_n) \leftarrow \mathsf{zip}_n\, e_1\ \ldots\ e_n\ ,\ e_b\ ]$.

The *work* of evaluating $\{\ e\ :\ x_1 \,\mathsf{in}\, e_1\,;\ldots;\, x_n \,\mathsf{in}\, e_n\ \mid\ e_b\ \}$ is sum of the cost of evaluating $e_1,\ldots,e_{n-1}$ and e_n plus the sum of the cost of evaluating e_b and e with the successive bindings to the elements of the results of the evaluation of $e_1,\ldots,e_n$. The *depth* of the evaluation is sum of the cost of evaluating $e_1,\ldots,e_{n-1}$ and e_n plus the maximum of the cost of evaluating e_b and e with the successive bindings to the elements of the results of the e_i.

Function Name / Function Type	Computed Depth Bound / Computed Work Bound	Run Time	Asym. Behav.
dyad	$10m + 10n + 3$	0.19 s	$O(n+m)$
$L(\text{int})*L(\text{int}) \to L(L(\text{int}))$	$10mn + 17n + 3$	0.20 s	$O(nm)$
dyad_all	$1.\bar{6}n^3-4n^2+10nm+14.\bar{6}n+5$	1.66 s	$O(n^2+m)$
$L(L(\text{int})) \to L(L(L(\text{int})))$	$1.\bar{3}n^3+5n^2m^2+8.5n^2m+\ldots$	0.96 s	$O(n^3+n^2m^2)$
m_mult1	$15xy + 16x + 10n + 6$	0.37 s	$O(xy)$
$L(L(\text{int}))*L(L(\text{int})) \to L(L(\text{int}))$	$15xyn + 16nm + 18n + 3$	0.36 s	$O(xyn)$
m_mult_pairs $[M := L(L(\text{int}))]$	$4n^2+15nmx+10nm+10n+3$	3.90 s	O(nm + mx)
$L(M)*L(M) \to L(M)$	$7.5n^2m^2x+7n^2m^2+n^2mx\ldots$	6.35 s	$O(n^2m^2x)$
m_mult2 $[M := L(L(\text{int}))]$	$35u + 10y + 15x + 11n + 40$	2.75 s	$O(z+x+n)$
$(M*\text{nat})*(M*\text{nat})\to M$	$3.5u^2y+uyz+14.5uy+\ldots$	2.99 s	$O(nx(z+y))$
quicksort_list	$12n^2 + 16nm + 12n + 3$	0.67 s	$O(n^2+m)$
$L(L(\text{int})) \to L(L(\text{int}))$	$8n^2m+15.5n^2-8nm+13.5n+3$	0.51 s	$O(n^2m)$
intersection	$10m + 12n + 3$	0.49 s	$O(n+m)$
$L(\text{int})*L(\text{int}) \to L(\text{int})$	$10mn + 19n + 3$	0.28 s	$O(nm)$
product	$8mn + 10m + 14n + 3$	1.05 s	$O(nm)$
$L(\text{int})*L(\text{int}) \to L(\text{int}*\text{int})$	$18mn + 21n + 3$	0.71 s	$O(nm)$
max_weight	$46n + 44$	0.39 s	$O(n)$
$L(\text{int}) \to \text{int}*L(\text{int})$	$13.5n^2 + 65.5n + 19$	0.30 s	$O(n^2)$
fib	$13n + 4$	0.09 s	$O(n)$
$\text{nat} * \text{nat} \to \text{nat}$	$- - -$	0.12 s	$O(2^n)$
dyad_comp	13	0.28 s	$O(1)$
$L(\text{int})*L(\text{int}) \to L(L(\text{int}))$	$6mn + 5n + 2$	0.13 s	$O(nm)$
find	$12m + 29n + 22$	0.38 s	$O(m+n)$
$L(\text{int})*L(\text{int}) \to L(L(\text{int}))$	$20mn + 18m + 9n + 16$	0.41 s	$O(nm)$

Table 1. Compilation of Computed Depth and Work Bounds.

7 Experimental Evaluation

We implemented the developed automatic depth analysis in Resource Aware ML (RAML). The implementation consists mainly of adding the syntactic form for the parallel binding and the parallel list comprehensions together with the treatment in the parser, the interpreter, and the resource-aware type system. RAML is publically available for download and through a user-friendly online interface [16]. On the project web page you also find the source code of all example programs and of RAML itself.

We used the implementation to perform an experimental evaluation of the analysis on typical examples from functional programming. In the compilation of our results we focus on examples that have a different asymptotic worst-case behavior in parallel and sequential evaluation. In many other cases, the worst-case behavior only differs in the constant factors. Also note that many of the classic examples of Blelloch [10]—like quick sort—have a better asymptotic average behavior in parallel evaluation but the same asymptotic worst-case behavior in parallel and sequential cost.

Table 1 contains a representative compilation of our experimental results. For each analyzed function, it shows the function type, the computed bounds on the work and the depth, the run time of the analysis in seconds and the actual asymptotic behavior of the function. The experiments were performed on an iMac with a 3.4 GHz Intel Core i7 and 8 GB memory. As LP solver we used IBM's

CPLEX and the constraint solving takes about 60% of the overall run time of the prototype on average. The computed bounds are simplified multivariate resource polynomials that are presented to the user by RAML. Note that RAML also outputs the (unsimplified) multivariate resource polynomials. The variables in the computed bounds correspond to the sizes of different parts of the input. As naming convention we use the order n, m, x, y, z, u of variables to name the sizes in a depth-first way: n is the size of the first argument, m is the maximal size of the elements of the first argument, x is the size of the second argument, etc.

All bounds are asymptotically tight if the tight bound is representable by a multivariate resource polynomial. For example, the exponential work bound for fib and the logarithmic bounds for bitonic_sort are not representable as a resource polynomial. Another example is the loose depth bound for dyad_all where we would need the base function $\max_{1 \leqslant i \leqslant n} m_i$ but only have $\sum_{1 \leqslant i \leqslant n} m_i$.

Matrix Operations. To study programs that use nested data structures we implemented several matrix operations for matrices that are represented by lists of lists of integers. The implemented operations include, the dyadic product from Section 3 (dyad), transposition of matrices (transpose, see [16]), addition of matrices (m_add, see [16]), and multiplication of matrices (m_mult1 and m_mult2).

To demonstrate the compositionality of the analysis, we have implemented two more involved functions for matrices. The function dyad_all computes the dyadic product (using dyad) of all ordered pairs of the inner lists in the argument. The function m_mult_pairs computes the products $M_1 \cdot M_2$ (using m_mult1) of all pairs of matrices such that M_1 is in the first list of the argument and M_2 is in the second list of the argument.

Sorting Algorithms. The sorting algorithms that we implemented include quick sort and bitonic sort for lists of integers (quicksort and bitonic_sort, see [16]).

The analysis computes asymptotically tight quadratic bounds for the work and depth of quick sort. The asymptotically tight bounds for the work and depth of bitonic sort are $O(n \log n)$ and $O(n \log^2 n)$, respectively, and can thus not be expressed by polynomials. However, the analysis computes quadratic and cubic bounds that are asymptotically optimal if we only consider polynomial bounds.

More interesting are sorting algorithms for lists of lists, where the comparisons need linear instead of constant time. In these algorithms we can often perform the comparisons in parallel. For instance, the analysis computes asymptotically tight bounds for quick sort for lists of lists of integers (quicksort_list, see Table 1).

Set Operations. We implemented sets as unsorted lists without duplicates. Most list operations such as intersection (Table 1), difference (see [16]), and union (see [16]) have linear depth and quadratic work. The analysis finds these asymptotically tight bounds.

The function product computes the Cartesian product of two sets. Work and depth of product are both linear and the analysis finds asymptotically tight bounds. However, the constant factors in the parallel evaluation are much smaller.

Miscellaneous. The function max_weight (Table 1) computes the maximal weight of a (connected) sublist of an integer list. The weight of a list is simply the sum of its elements. The work of the algorithm is quadratic but the depth is linear.

Finally, there is a large class of programs that have non-polynomial work but polynomial depth. Since the analysis can only compute polynomial bounds we can only derive bounds on the depth for such programs. A simple example in Table 1 is the function fib that computes the Fibonacci numbers without memoization.

Parallel List Comprehensions. The aforementioned examples are all implemented without using parallel list comprehensions. Parallel list comprehensions have a better asymptotic behavior than semantically-equivalent recursive functions in RAML's current resource metric for evaluation steps.

A simple example is the function dyad_comp which is equivalent to dyad and which is implemented with the expression $\{\{x * y : y \text{ in } ys\} : x \text{ in } xs\}$. As listed in Table 1, the depth of dyad_comp is constant while the depth of dyad is linear. RAML computes tight bounds.

A more involved example is the function find that finds a given integer list (needle) in another list (haystack). It returns the starting indices of each occurrence of the needle in the haystack. The algorithm is described by Blelloch [15] and cleverly uses parallel list comprehensions to perform the search in parallel. RAML computes asymptotically tight bounds on the work and depth.

Discussion. Our experiments show that the range of the analysis is not reduced when deriving bounds on the depth: The prototype implementation can always infer bounds on the depth of a program if it can infer bounds on the sequential version of the program. The derivation of bounds for parallel programs is also almost as efficient as the derivation of bounds for sequential programs.

We experimentally compared the derived worst-case bounds with the measured work and depth of evaluations with different inputs. In most cases, the derived bounds on the depth are asymptotically tight and the constant factors are close or equal to the optimal ones. As a representative example, the full version of the article contains plots of our experiments for quick sort for lists of lists.

8 Related Work

Automatic amortized resource analysis was introduced by Hofmann and Jost for a strict first-order functional language [3]. The technique has been applied to higher-order functional programs [22], to derive stack-space bounds for functional programs [23], to functional programs with lazy evaluation [4], to object-oriented programs [24, 25], and to low-level code by integrating it with separation logic [26]. All the aforementioned amortized-analysis–based systems are limited to linear bounds. The polynomial potential functions that we use in this paper were introduced by Hoffmann et al. [19, 13, 7]. In contrast to this work, none of the previous works on amortized analysis considered parallel evaluation. The main technical innovation of this work is the new rule for parallel composition that is not straightforward. The smooth integration of this rule in the existing framework of multivariate amortized resource analysis is a main advantages of our work.

Type systems for inferring and verifying cost bounds for sequential programs have been extensively studied. Vasconcelos et al. [27, 1] described an automatic analysis system that is based on sized-types [28] and derives linear bounds for

higher-order sequential functional programs. Dal Lago et al. [29, 30] introduced linear dependent types to obtain a complete analysis system for the time complexity of the call-by-name and call-by-value lambda calculus. Crary and Weirich [31] presented a type system for specifying and certifying resource consumption. Danielsson [32] developed a library, based on dependent types and manual cost annotations, that can be used for complexity analyses of functional programs. We are not aware of any type-based analysis systems for parallel evaluation.

Classically, cost analyses are often based on deriving and solving recurrence relations. This approach was pioneered by Wegbreit [33] and has been extensively studied for sequential programs written in imperative languages [6, 34] and functional languages [35, 2].

In comparison, there has been little work done on the analysis of parallel programs. Albert et al. [36] use recurrence relations to derive cost bounds for concurrent object-oriented programs. Their model of concurrent imperative programs that communicate over a shared memory and the used cost measure is however quite different from the depth of functional programs that we study.

The only article on using recurrence relations for deriving bounds on parallel functional programs that we are aware of is a technical report by Zimmermann [37]. The programs that were analyzed in this work are fairly simple and more involved programs such as sorting algorithms seem to be beyond its scope. Additionally, the technique does not provide the compositionality of amortized resource analysis.

Trinder et al. [38] give a survey of resource analysis techniques for parallel and distributed systems. However, they focus on the usage of analyses for sequential programs to improve the coordination in parallel systems. Abstract interpretation based approaches to resource analysis [5, 39] are limited to sequential programs.

Finally, there exists research that studies cost models to formally analyze parallel programs. Blelloch and Greiner [10] pioneered the cost measures work and depth that we use in this work. There are more advanced cost models that take into account caches and IO (see, e.g., Blelloch and Harper [11]), However, these works do not provide machine support for deriving static cost bounds.

9 Conclusion

We have introduced the first type-based cost analysis for deriving bounds on the depth of evaluations of parallel function programs. The derived bounds are multivariate resource polynomials that can express a wide range of relations between different parts of the input. As any type system, the analysis is naturally compositional. The new analysis system has been implemented in Resource Aware ML (RAML) [14]. We have performed a thorough and reproducible experimental evaluation with typical examples from functional programming that shows the practicability of the approach.

An extension of amortized resource analysis to handle non-polynomial bounds such as max and log in a compositional way is an orthogonal research question that we plan to address in the future. A promising direction that we are currently studying is the use of numerical logical variables to guide the analysis to derive non-polynomial bounds. The logical variables would be treated like regular

variables in the analysis. However, the user would be responsible for maintaining and proving relations such as $a = \log n$ where a is a logical variable an n is the size of a regular data structure. In this way, we would gain flexibility while maintaining the compositionality of the analysis.

Another orthogonal question is the extension of the analysis to additional language features such as higher-order functions, references, and user-defined data structures. These extensions have already been implemented in a prototype and pose interesting research challenges in there own right. We plan to report on them in a forthcoming article.

Acknowledgments. This research is based on work supported in part by NSF grants 1319671 and 1065451, and DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Vasconcelos, P.: Space Cost Analysis Using Sized Types. PhD thesis, School of Computer Science, University of St Andrews (2008)
2. Danner, N., Paykin, J., Royer, J.S.: A Static Cost Analysis for a Higher-Order Language. In: 7th Workshop on Prog. Languages Meets Prog. Verification (PLPV'13). (2013) 25–34
3. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-Order Functional Programs. In: 30th ACM Symp. on Principles of Prog. Langs. (POPL'03). (2003) 185–197
4. Simões, H.R., Vasconcelos, P.B., Florido, M., Jost, S., Hammond, K.: Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In: 17th Int. Conf. on Funct. Prog. (ICFP'12). (2012) 165–176
5. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: 36th ACM Symp. on Principles of Prog. Langs. (POPL'09). (2009) 127–139
6. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: 16th Euro. Symp. on Prog. (ESOP'07). (2007) 157–172
7. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. ACM Trans. Program. Lang. Syst. (2012)
8. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating Runtime and Size Complexity Analysis of Integer Programs. In: Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14). (2014) 140–155
9. Sinn, M., Zuleger, F., Veith, H.: A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In: Computer Aided Verification - 26th Int. Conf. (CAV'14). (2014) 743–759
10. Blelloch, G.E., Greiner, J.: A Provable Time and Space Efficient Implementation of NESL. In: 1st Int. Conf. on Funct. Prog. (ICFP'96). (1996) 213–225
11. Blelloch, G.E., Harper, R.: Cache and I/O Effcent Functional Algorithms. In: 40th ACM Symp. on Principles Prog. Langs. (POPL'13). (2013) 39–50
12. Harper, R.: Practical Foundations for Programming Languages. Cambridge University Press (2012)
13. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate Amortized Resource Analysis. In: 38th ACM Symp. on Principles of Prog. Langs. (POPL'11). (2011)

14. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource Aware ML. In: 24rd Int. Conf. on Computer Aided Verification (CAV'12). (2012)
15. Blelloch, G.E.: Nesl: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, CMU (1995)
16. Aehlig, K., Hofmann, M., Hoffmann, J.: RAML Web Site. `http://raml.tcs.ifi.lmu.de` (2010-2014)
17. Hoffmann, J., Shao, Z.: Automatic Static Cost Analysis for Parallel Programs. `http://cs.yale.edu/~hoffmann/papers/parallelcost2014.pdf` (2014) Full Version.
18. Charguéraud, A.: Pretty-Big-Step Semantics. In: 22nd Euro. Symp. on Prog. (ESOP'13). (2013) 41–60
19. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: 19th Euro, Symp. on Prog. (ESOP'10). (2010)
20. Hoffmann, J., Shao, Z.: Type-Based Amortized Resource Analysis with Integers and Arrays. In: 12th International Symposium on Functional and Logic Programming (FLOPS'14). (2014)
21. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In: Prog. Langs. and Systems - 8th Asian Symposium (APLAS'10). (2010)
22. Jost, S., Hammond, K., Loidl, H.W., Hofmann, M.: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: 37th ACM Symp. on Principles of Prog. Langs. (POPL'10). (2010) 223–236
23. Campbell, B.: Amortised Memory Analysis using the Depth of Data Structures. In: 18th Euro. Symp. on Prog. (ESOP'09). (2009) 190–204
24. Hofmann, M., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: 15th Euro. Symp. on Prog. (ESOP'06). (2006) 22–37
25. Hofmann, M., Rodriguez, D.: Automatic Type Inference for Amortised Heap-Space Analysis. In: 22nd Euro. Symp. on Prog. (ESOP'13). (2013) 593–613
26. Atkey, R.: Amortised Resource Analysis with Separation Logic. In: 19th Euro. Symp. on Prog. (ESOP'10). (2010) 85–103
27. Vasconcelos, P.B., Hammond, K.: Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In: Int. Workshop on Impl. of Funct. Langs. (IFL'03), Springer-Verlag LNCS (2003) 86–101
28. Hughes, J., Pareto, L., Sabry, A.: Proving the Correctness of Reactive Systems Using Sized Types. In: 23th ACM Symp. on Principles of Prog. Langs. (POPL'96). (1996) 410–423
29. Lago, U.D., Gaboardi, M.: Linear Dependent Types and Relative Completeness. In: 26th IEEE Symp. on Logic in Computer Science (LICS'11). (2011) 133–142
30. Lago, U.D., Petit, B.: The Geometry of Types. In: 40th ACM Symp. on Principles Prog. Langs. (POPL'13). (2013) 167–178
31. Crary, K., Weirich, S.: Resource Bound Certification. In: 27th ACM Symp. on Principles of Prog. Langs. (POPL'00). (2000) 184–198
32. Danielsson, N.A.: Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In: 35th ACM Symp. on Principles Prog. Langs. (POPL'08). (2008) 133–144
33. Wegbreit, B.: Mechanical Program Analysis. Commun. ACM **18**(9) (1975) 528–539
34. Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: 19th Int. Static Analysis Symposium (SAS'12). (2012) 405–421
35. Grobauer, B.: Cost Recurrences for DML Programs. In: 6th Int. Conf. on Funct. Prog. (ICFP'01). (2001) 253–264

36. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Prog. Langs. and Systems - 9th Asian Symposium (APLAS'11). (2011) 238–254
37. Zimmermann, W.: Automatic Worst Case Complexity Analysis of Parallel Programs. Technical Report TR-90-066, University of California, Berkeley (1990)
38. Trinder, P.W., Cole, M.I., Hammond, K., Loidl, H.W., Michaelson, G.: Resource Analyses for Parallel and Distributed Coordination. Concurrency and Computation: Practice and Experience **25**(3) (2013) 309–348
39. Zuleger, F., Sinn, M., Gulwani, S., Veith, H.: Bound Analysis of Imperative Programs with the Size-change Abstraction. In: 18th Int. Static Analysis Symposium (SAS'11). (2011)

Compositional Certified Resource Bounds

Quentin Carbonneaux Jan Hoffmann Zhong Shao
Yale University, USA
{quentin.carbonneaux, jan.hoffmann, zhong.shao}@yale.edu

Abstract

This paper presents a new approach for automatically deriving worst-case resource bounds for C programs. The described technique combines ideas from amortized analysis and abstract interpretation in a unified framework to address four challenges for state-of-the-art techniques: compositionality, user interaction, generation of proof certificates, and scalability. *Compositionality* is achieved by incorporating the potential method of amortized analysis. It enables the derivation of global whole-program bounds with local derivation rules by naturally tracking size changes of variables in sequenced loops and function calls. The resource consumption of functions is described abstractly and a function call can be analyzed without access to the function body. *User interaction* is supported with a new mechanism that clearly separates qualitative and quantitative verification. A user can guide the analysis to derive complex non-linear bounds by using auxiliary variables and assertions. The assertions are separately proved using established qualitative techniques such as abstract interpretation or Hoare logic. *Proof certificates* are automatically generated from the local derivation rules. A soundness proof of the derivation system with respect to a formal cost semantics guarantees the validity of the certificates. *Scalability* is attained by an efficient reduction of bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. The analysis framework is implemented in the publicly-available tool C^4B. An experimental evaluation demonstrates the advantages of the new technique with a comparison of C^4B with existing tools on challenging micro benchmarks and the analysis of more than 2900 lines of C code from the cBench benchmark suite.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability

Keywords Quantitative Verification, Resource Bound Analysis, Static Analysis, Amortized Analysis, LP Solving, Program Logic

PLDI'15, June 13–17, 2015, Portland, OR, USA.

ACM 978-1-4503-3468-6/15/06. . . $15.00.
http://dx.doi.org/10.1145/2737924.2737955

1. Introduction

In software engineering and software verification, we often would like to have static information about the quantitative behavior of programs. For example, stack and heap-space bounds are important to ensure the reliability of safety-critical systems [37]. Static energy usage information is critical for autonomous systems and has applications in cloud computing [17, 18]. Worst-case time bounds can help create constant-time implementations that prevent side-channel attacks [9, 32]. Loop and recursion-depth bounds are used to ensure the accuracy of programs that are executed on unreliable hardware [14] and complexity bounds are needed to verify cryptographic protocols [8]. In general, quantitative resource information can provide useful feedback for developers.

Available techniques for *automatically* deriving worst-case resource bounds fall into two categories. Techniques in the first category derive impressive bounds for numerical imperative programs, but are not compositional. This is problematic if one needs to derive global whole-program bounds. Techniques in the second category derive tight whole-program bounds for programs with regular loop or recursion patterns that decrease the size of an individual variable or data structure. They are highly compositional, scale for large programs, and work directly on the syntax. However, they do not support multivariate interval-based resource bounds (e.g., $x - y$) which are common in C programs. Indeed, it has been a long-time open problem to develop compositional resource analysis techniques that can work for typical imperative code with non-regular iteration patterns, signed integers, mutation, and non-linear control flow.

Tools in the first category include SPEED [22], KoAT [13], PUBS [1], Rank [3], and LOOPUS [38]. They lack compositionality in at least two ways. First, they all base their analysis on some form of *ranking function* or *counter instrumentation* that is linked to a local analysis. As a result, loop bounds are arithmetic expressions that depend on the values of variables just before the loop. This makes it hard to give a resource bound on a sequence of loops and function calls in terms of the input parameters of a function. Second, while all popular imperative programming languages provide a function or procedure abstraction, available tools are not able to abstract resource behavior; instead, they have to inline the procedure body to perform their analysis.

Tools in the second category originate form the *potential method* of amortized analysis and type systems for functional programs [26, 28]. It has been shown that class definitions of object-oriented programs [29] and data-structure predicates of separation logic [7] can play the role of the type system in imperative programs. However, a major weakness of existing potential-based techniques is that they can only associate *potential* with individual program variables or data structures. For C programs, this fails for loops as simple as `for(i=x;i<y;i++)` where $y - i$ decreases, but not $|i|$.

A general problem with existing tools (in both categories) is user interaction. When a tool fails to find a resource bound for a program, there is no possibility for sound user interaction to guide

the tool during bound derivation. For example, there is no concept of manual proofs of resource bounds; and no framework can support composition of manually derived bounds with automatically inferred bounds.

This paper presents a new compositional framework for automatically deriving resource bounds on C programs. This new approach is an attempt to unify the two aforementioned categories: It solves the compositionality issues of techniques for numerical imperative code by adapting amortized-analysis–based techniques from the functional world. Our automated analysis is able to infer resource bounds on C programs with mutually-recursive functions and integer loops. The resource behavior of functions can be summarized in a functional specification that can be used at every call site without accessing the function body. To our knowledge this is the first technique based on amortized analysis that is able to derive bounds that depend on negative numbers and differences of variables. It is also the first resource analysis technique for C that deals naturally with recursive functions and sequenced loops, and can handle resources that may become available during execution (e.g., when freeing memory). Compared to more classical approaches based on ranking functions, our tool inherits the benefits of amortized reasoning. Using only one simple mechanism, it handles:

- interactions between *sequential loops or function calls* through size changes of variables,
- *nested loops* that influence each other with the same set of modified variables,
- and *amortized bounds* as found, for example, in the Knuth-Morris-Pratt algorithm for string search.

The main innovations that make amortized analysis work on imperative languages are to base the analysis on a Hoare-like logic and to track multivariate quantities instead of program variables. This leads to precise bounds expressed as functions of sizes $|[x, y]| = \max(0, y - x)$ of intervals. A distinctive feature of our analysis system is that it reduces linear bound inference to a linear optimization problem that can be solved by off-the-shelf LP solvers. This enables the efficient inference of global bounds for larger programs. Moreover, our local inference rules automatically generate proof certificates that can be easily checked in linear time.

The use of the potential method of amortized analysis makes user interaction possible in different ways. For one thing, we can directly combine the new automatic analysis with manually derived bounds in a previously-developed quantitative Hoare logic [15] (see Section 7). For another thing, we describe a new mechanism that allows the separation of quantitative and qualitative verification (see Section 6). Using this mechanism, the user can guide the analysis by using auxiliary variables and logical assertions that can be verified by existing qualitative tools such as Hoare logic or abstract interpretation. In this way, we can benefit from existing automation techniques and provide a middle-ground between fully automatic and fully manual verification for bound derivation. This enables the semi-automatic inference of non-linear bounds, such as polynomial, logarithmic, and exponential bounds.

We have implemented the analysis system in the tool C^4B and experimentally evaluated its effectiveness by analyzing system code and examples from the literature. C^4B has automatically derived global resource bounds for more than 2900 lines of C code from the cBench benchmark suite. The extended version of this article [16] contains more than 30 challenging loop and recursion patterns that we collected from open source software and the literature. Our analysis can find asymptotically tight bounds for all but one of these patterns, and in most cases the derived constant factors are tight. To compare C^4B with existing techniques, we tested our examples with tools such as KoAT [13], Rank [3], and LOOPUS [38]. Our experiments show that the bounds that we derive are often more precise than those derived by existing tools. Only LOOPUS [38], which also uses amortization techniques, is able to achieve a similar precision.

Examples from cBench and micro benchmarks demonstrate the practicality and expressiveness of the user guided bound inference. For example, we derive a logarithmic bound for a binary search function and a bound that amortizes the cost of k increments to a binary counter (see Section 6).

In summary, we make the following *contributions*.

- We develop the first automatic amortized analysis for C programs. It is naturally compositional, tracks size changes of variables to derive global bounds, can handle mutually-recursive functions, generates resource abstractions for functions, derives proof certificates, and handles resources that may become available during execution.
- We show how to automatically reduce the inference of *linear* resource bounds to efficient LP solving.
- We describe a new method of harnessing existing qualitative verification techniques to guide the automatic amortized analysis to derive *non-linear* resource bounds with LP solving.
- We prove the soundness of the analysis with respect to a parametric cost semantics for C programs. The cost model can be further customized with function calls ($\mathsf{tick(n)}$) that indicate resource usage.
- We implemented our resource bound analysis in the publicly-available tool C^4B.
- We present experiments with C^4B on more than 2900 lines of C code. A detailed comparison shows that our prototype is the only tool that can derive global bounds for larger C programs while being as powerful as existing tools when deriving linear local bounds for tricky loop and recursion patterns.

2. The Potential Method

The idea that underlies the design of our framework is amortized analysis [39]. Assume that a program S executes on a starting state σ and consumes n resource units of some user-defined quantity. We denote that by writing $(S, \sigma) \Downarrow_n \sigma'$ where σ' is the program state after the execution. The basic idea of amortized analysis is to define a *potential function* Φ that maps program states to non-negative numbers and to show that $\Phi(\sigma) \geqslant n$ if σ is a program state such that $(S, \sigma) \Downarrow_n \sigma'$. Then $\Phi(\sigma)$ is a valid resource bound.

To obtain a compositional reasoning we also have to take into account the state resulting from a program's execution. We thus use two potential functions, one that applies before the execution, and one that applies after. The two functions must respect the relation $\Phi(\sigma) \geqslant n + \Phi'(\sigma')$ for all states σ and σ' such that $(S, \sigma) \Downarrow_n \sigma'$. Intuitively, $\Phi(\sigma)$ must provide enough *potential* for both, paying for the resource cost of the computation and paying for the potential $\Phi'(\sigma')$ on the resulting state σ'. That way, if $(\sigma, S_1) \Downarrow_n \sigma'$ and $(\sigma', S_2) \Downarrow_m \sigma''$, we get $\Phi(\sigma) \geqslant n + \Phi'(\sigma')$ and $\Phi'(\sigma') \geqslant m + \Phi''(\sigma'')$. This can be composed as $\Phi(\sigma) \geqslant (n+m) + \Phi''(\sigma'')$. Note that the initial potential function Φ provides an upper bound on the resource consumption of the whole program. What we have observed is that, if we define $\{\Phi\}\, S\, \{\Phi'\}$ to mean

$$\forall \sigma\, n\, \sigma'.\, (\sigma, S) \Downarrow_n \sigma' \implies \Phi(\sigma) \geqslant n + \Phi'(\sigma')\ ,$$

then we get the following familiar looking rule

$$\frac{\{\Phi\}\, S_1\, \{\Phi'\} \qquad \{\Phi'\}\, S_2\, \{\Phi''\}}{\{\Phi\}\, S_1; S_2\, \{\Phi''\}}\ .$$

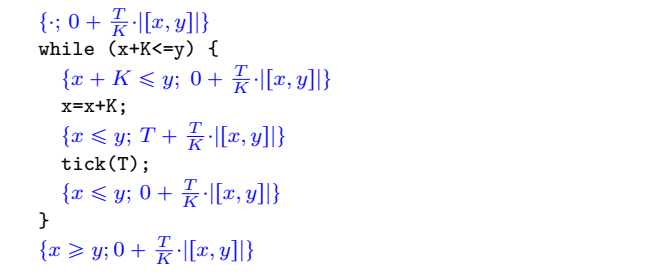


Figure 1. Derivation of a tight bound on the number of ticks for a standard *for loop*. The parameters $K > 0$ and $T > 0$ are not program variables but denote concrete constants.

This rule already shows a departure from classical techniques that are based on ranking functions. Reasoning with two potential functions promotes compositional reasoning by focusing on the sequencing of programs. In the previous rule, Φ gives a bound for $S_1; S_2$ through the intermediate potential Φ', even though it was derived on S_1 only. Similarly, other language constructs lead to rules for the potential functions that look very similar to Hoare logic or effect system rules. These rules enable reasoning about resource usage in a flexible and compositional way, which, as a side effect, produces a certificate for the derived resource bound.

The derivation of a resource bound using potential functions is best explained by example. If we use the tick metric that assigns cost n to the function call $\mathsf{tick(n)}$ and cost 0 to all other operations then the cost of the following example can be bounded by $|[x, y]| = \max(y-x, 0)$.

```
while (x<y) { x=x+1; tick(1); }
```
(Example 1)

To derive this bound, we start with the initial potential $\Phi_0 = |[x, y]|$, which we also use as the loop invariant. For the loop body we have (like in Hoare logic) to derive a triple $\{\Phi_0\}\, \mathsf{x = x + 1; tick(1)}\, \{\Phi_0\}$. We can only do so if we utilize the fact that $x < y$ at the beginning of the loop body. The reasoning then works as follows. We start with the potential $|[x, y]|$ and the fact that $|[x, y]| > 0$ before the assignment. If we denote the updated version of x after the assignment by x' then the relation $|[x, y]| = |[x', y]| + 1$ between the potential before and after the assignment $\mathsf{x = x + 1}$ holds. This means that we have the potential $|[x, y]| + 1$ before the statement $\mathsf{tick(1)}$. Since $\mathsf{tick(1)}$ consumes one resource unit, we end up with potential $|[x, y]|$ after the loop body and have established the loop invariant again.

Figure 1 shows a derivation of the bound $\frac{T}{K}\cdot|[x, y]|$ on the number of ticks for a generalized version of Example 1 in which we increment x by a constant $K > 0$ and consume $T > 0$ resources in each iteration. The reasoning is similar to the one of Example 1 except that we obtain the potential $K \cdot \frac{T}{K}$ after the assignment. In the figure, we separate logical assertions from potential functions with semicolons. Note that the logical assertions are only used in the rule for the assignment $\mathsf{x = x + K}$.

To the best of our knowledge, no other implemented tool for C is currently capable of deriving a tight bound on the cost of such a loop. For $T = 1$ (many systems focus on the number of loop iterations without a cost model) and $K = 10$, KoAT computes the bound $|x| + |y| + 10$, Rank computes the bound $y - x - 7$, and LOOPUS computes the bound $y-x-9$. Only PUBS computes the tight bound $0.1(y - x)$ if we translate the program into a term-rewriting system by hand. We will show in the following sections that the potential method makes automatic bound derivation straightforward.

The concept of a potential function is a generalization of the concept of a ranking function. A potential function can be used like a ranking function if we use the tick metric and add the statement $\mathsf{tick(1)}$ to every back edge of the program (loops and function calls). However, a potential function is more flexible. For example, we can use a potential function to prove that Example 2 does not consume any resources in the tick metric.

```
while (x<y) {tick(-1); x=x+1; tick(1)}
```
(Example 2)

```
while (x<y) { x=x+1; tick(10); }
```
(Example 3)

Similarly we can prove that Example 3 can be bounded by $10|[x, y]|$. In both cases, we reason exactly like in the first version of the while loop to prove the bound. Of course, such loops with different tick annotations can be seamlessly combined in a larger program.

3. Compositional Resource-Bound Analysis

In this section we describe the high-level design of the automatic amortized analysis that we implemented in C^4B. Examples explain and motivate our design decisions.

Linear Potential Functions. To find resource bounds automatically, we first need to restrict our search space. In this work, we focus on the following form of potential functions, which can express tight bounds for many typical programs and allows for inference with *linear programming*.

$$\Phi(\sigma) = q_0 + \sum_{x,y \in \mathrm{dom}(\sigma) \,\wedge\, x \neq y} q_{(x,y)} \cdot |[\sigma(x), \sigma(y)]|\,.$$

Here $\sigma : (\mathrm{Locals} \to \mathbb{Z}) \times (\mathrm{Globals} \to \mathbb{Z})$ is a simplified program state that maps variable names to integers, $|[a, b]| = \max(0, b - a)$, and $q_i \in \mathbb{Q}_0^+$. To simplify the references to the linear coefficients q_i, we introduce an *index set* I. This set is defined to be $\{0\} \cup \{(x, y) \mid x, y \in \mathit{Var} \wedge x \neq y\}$. Each index i corresponds to a *base function* f_i in the potential function: 0 corresponds to the constant function $\sigma \mapsto 1$, and (x, y) corresponds to $\sigma \mapsto |[\sigma(x), \sigma(y)]|$. Using these notations we can rewrite the above equality as $\Phi(\sigma) = \sum_{i \in I} q_i f_i(\sigma)$.We often write xy to denote the index (x, y). This allows us to uniquely represent any linear potential function Φ as a *quantitative annotation* $Q = (q_i)_{i \in I}$, that is, a family of non-negative rational numbers where only a finite number of elements are not zero.

In the potential functions, we treat constants as global variables that cannot be assigned to. For example, if the program contains the constant 1988 then we have a variable c_{1988} and $\sigma(c_{1988}) = 1988$. We assume that every program state includes the constant c_0.

Abstract Program State. In addition to the quantitative annotations, our automatic amortized analysis needs to maintain a minimal abstract state to justify certain operations on quantitative annotations. For example when analyzing the code $x \leftarrow x + y$, it is helpful to know the sign of y to determine which intervals will increase or decrease. The knowledge needed by our rules can be inferred by local reasoning (i.e. in basic blocks without recursion and loops) within usual theories (e.g. Presburger arithmetic or bit vectors).

The abstract program state is represented as *logical contexts* in the derivation system used by our automated tool. Our implementation finds these logical contexts using abstract interpretation with the domain of linear inequalities. We observed that the rules of the analysis often require only minimal local knowledge. This means that it is not necessary for us to compute precise loop invariants and only a rough fixpoint (e.g. keeping only inequalities on variables unchanged by the loop) is sufficient to obtain good bounds.

Challenging Loops. One might think that our set of potential functions is too simplistic to be able to express and prove bounds for realistic programs. Nevertheless, we can handle challenging example programs without special tricks or techniques. Examples

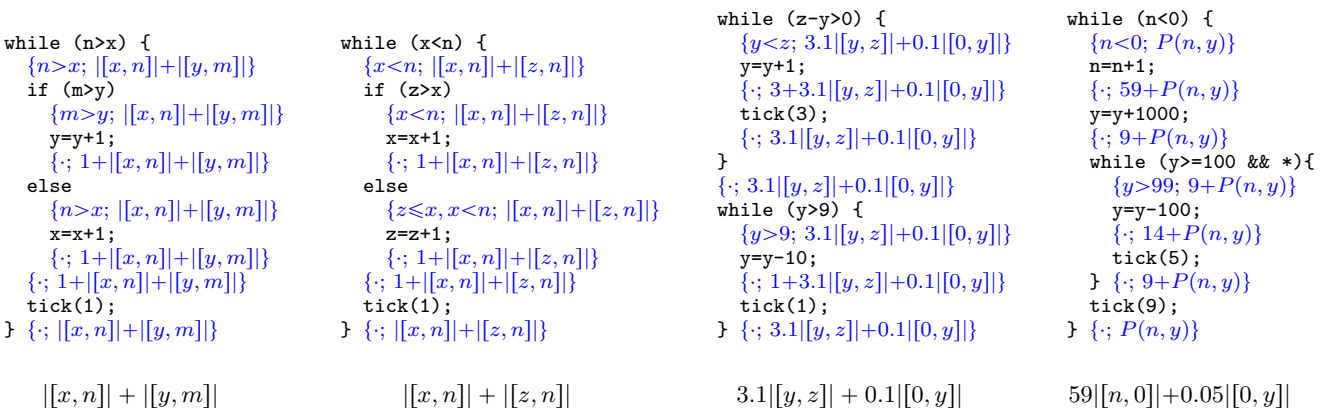


Figure 2. Derivations of bounds on the number of ticks for challenging examples. Examples *speed_1* and *speed_2* (from [22]) use *tricky iteration patterns*, *t08a* contains *sequential loops* so that the iterations of the second loop depend on the first, and *t27* contains interacting *nested loops*. In Example *t27*, we use the abbreviation $P(n,y) := 59|[n,0]|+0.05|[0,y]|$.

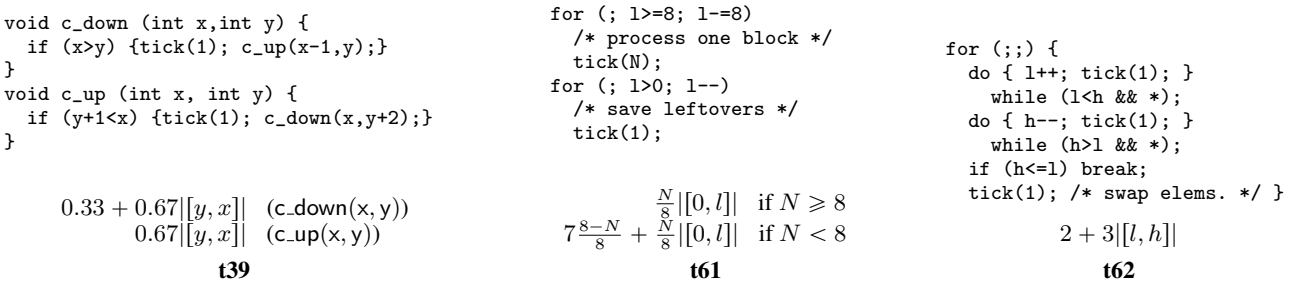


Figure 3. Example *t39* shows two mutually-recursive functions with the computed tick bounds. Example *t61* and *t62* demonstrate the unique compositionality of our system. In *t61*, $N \geqslant 0$ is a fixed but arbitrary constant.

speed_1 and *speed_2* in Figure 2, which are taken from previous work [22], demonstrate that our method can handle *tricky iteration patterns*. The SPEED tool [22] derives the same bounds as our analysis but requires heuristics for its counter instrumentation. These loops can also be handled with inference of *disjunctive invariants*, but in the abstract interpretation community, these invariants are known to be notoriously difficult to generate. In Example *speed_1* we have one loop that first increments variable y up to m and then increments variable x up to n. We derive the tight bound $|[x,n]| + |[y,m]|$. Example *speed_2* is even trickier, and we found it hard to find a bound manually. However, using potential transfer reasoning as in amortized analysis, it is easy to prove the tight bound $|[x,n]| + |[z,n]|$.

Nested and Sequenced Loops. Example *t08a* in Figure 2 shows the ability of the analysis to discover interaction between *sequenced loops* through size change of variables. We accurately track the size change of y in the first loop by transferring the potential 0.1 from $|[y,z]|$ to $|[0,y]|$. Furthermore, *t08a* shows again that we do not handle the constants 1 or 0 in any special way. In all examples we could replace 0 and 1 with other constants like in the second loop and still derive a tight bound. Example *t27* in Figure 2 shows how amortization can be used to handle *interacting nested loops*. In the outer loop we increment the variable n until $n = 0$. In each of the $|[n,0]|$ iterations, we increment the variable y by 1000. Then we non-deterministically (expressed by $*$) execute an inner loop that decrements y by 100 until $y < 100$. The analysis discovers that only the first execution of the inner loop depends on the initial value of y. We again derive tight constant factors.

Mutually Recursive Functions. As mentioned, the analysis also handles advanced control flow like `break` and `return` statements, and mutual recursion. Example *t39* in Figure 3 contains two mutually-recursive functions with their automatically derived tick bounds. The function `c_down` decrements its first argument x until it reaches the second argument y. It then recursively calls the function `c_up`, which is dual to `c_down`. Here, we count up y by 2 and call `c_down`. C^4B is the only available system that computes a tight bound.

Compositionality. With two concrete examples from open-source projects we demonstrate that the compositionality of our method is indeed crucial in practice.

Example *t61* in Figure 3 is typical for implementations of block-based cryptographic primitives: Data of arbitrary length is consumed in blocks and the leftover is stored in a buffer for future use when more data is available. It is present in all the block encryption routines of PGP and also used in performance critical code to unroll a loop. For example we found it in a bit manipulating function of the libtiff library and a CRC computation routine of MAD, an MPEG decoder. This looping pattern is handled particularly well by our method. If $N \geqslant 8$, C^4B infers the bound $\frac{N}{8}|[0,l]|$, but if $N < 8$, it infers $7\frac{8-N}{8} + \frac{N}{8}|[0,l]|$. The selection of the block size (8) and the cost in the second loop (`tick(1)`) are random choices and C^4B would also derive tight bound for other values.

To understand the resource bound for the case $N < 8$, first note that the cost of the second loop is $|[0, l]|$. After the first loop, we still have $\frac{N}{8}|[0, l]|$ potential available from the invariant. So we have to raise the potential of $|[0, l]|$ from $\frac{N}{8}$ to 1, that is, we must pay $\frac{8-N}{8}|[0, l]|$. But since we got out of the first loop, we know that $l < 8$, so it is sound to only pay $7\frac{8-N}{8}$ potential units instead. This level of precision and compositionality is only achieved by our novel analysis, no other available tool derives the aforementioned tight bounds.

Example *t62* (Figure 3) is the inner loop of a quick sort implementation in cBench. More precisely, it is the partitioning part of the algorithm. This partition loop has linear complexity, and feeding it to our analysis gives the worst-case bound $2 + 3|[l, h]|$. This bound is not optimal but it can be refined by rewriting the program. To understand the bound, we can reason as follows. If $h \geqslant l$ initially, the cost of the loop is 2. Otherwise, the cost of each round (at most 3) can be payed using the potential of $[l, h]$ by the first increment to l because we know that $l < h$. The two inner loops can also use $[l, h]$ to pay for their inner costs. KoAT fails to find a bound and LOOPUS derives the quadratic bound $(h - l - 1)^2$. Following the classical technique, these tools try to find one ranking function for each loop and combine them multiplicatively or additively.

In the extended version [16] is a list of more than 30 classes of challenging programs that we can automatically analyze. Section 8 contains a more detailed comparison with other tools.

4. Derivation System

In the following we describe the local and compositional derivation rules of the automatic amortized analysis.

Cost Aware Clight. We present the rules for a subset of Clight. Clight is the first intermediate language of the CompCert compiler [34]. It is a subset of C with a unified looping construct and side-effect free expressions. We reuse most of CompCert's syntax but instrument the semantics with a resource metric M that accounts for the cost (an arbitrary rational number) of each step in the operational semantics. For example, $M_e(\mathsf{exp})$ is the cost of evaluating the expression exp. The rationals M_f and M_r account respectively for the cost of a call to the function f and the cost of returning from it. More details are provided in Section 7.

In the rules, assignments are restricted to the form $x \leftarrow y$ or $x \leftarrow x \pm y$. In the implementation, a Clight program is converted into this form prior to analysis without changing the resource cost. This is achieved by using a series of *cost-free assignments* that do not result in additional cost in the semantics. Non-linear operations such as $x \leftarrow z * y$ or $x \leftarrow a[y]$ are handled by assigning 0 to coefficients like q_{xa} and q_{ax} that contain x after the assignment. This sound treatment ensures that no further loop bounds depend on the result of the non-linear operation.

Judgements. The derivation system for the automatic amortized analysis is defined in Figure 4. The derivation rules derive judgements of the form

$$(\Gamma_B; Q_B), (\Gamma_R; Q_R) \vdash \{\Gamma; Q\}\, S\, \{\Gamma'; Q'\}.$$

The part $\{\Gamma; Q\}\, S\, \{\Gamma'; Q'\}$ of the judgement can be seen as a quantitative Hoare triple. All assertions are split into two parts, the logical part and the quantitative part. The quantitative part Q represents a potential function as a collection of non-negative numbers q_i indexed by the index set I. The logical part Γ is left abstract but is enforced by our derivation system to respect classic Hoare logic constraints. The meaning of this basic judgment is as follows: If S is executed with starting state σ, the assertions in Γ hold, and at least $Q(\sigma)$ resources are available then the evaluation does not run out of resources and, if the execution terminates in state σ', there are at least $Q'(\sigma')$ resources left and Γ' holds for σ'.

The judgement is a bit more involved since we have to take into account the early exit statements break and return. This is similar to classical Hoare triples in the presence of non-linear control flow. In the judgement, $(\Gamma_B; Q_B)$ is the postcondition that holds when breaking out of a loop using break. Similarly, $(\Gamma_R; Q_R)$ is the postcondition that holds when returning from a function call.

As a convention, if Q and Q' are quantitative annotations we assume that $Q = (q_i)_{i \in I}$ and $Q' = (q'_i)_{i \in I}$. The notation $Q \pm n$ used in many rules defines a new context Q' such that $q'_0 = q_0 \pm n$ and $\forall i \neq 0.\, q'_i = q_i$. In all the rules, we have the implicit side condition that all rational coefficients are non-negative. Finally, if a rule mentions Q and Q' and leaves the latter undefined at some index i we assume that $q'_i = q_i$.

Function Specifications. During the analysis, function specifications are quadruples $(\Gamma_f; Q_f, \Gamma'_f; Q'_f)$ where $\Gamma_f; Q_f$ depend on $a\vec{r}gs$, and $\Gamma'_f; Q'_f$ depend on ret. These parameters are instantiated by appropriate variables on call sites. A distinctive feature of our analysis is that it respects the function abstraction: when deriving a function specification it generates a set of constraints and the above quadruple; once done, the constraint set can readily be reused for every call site and the function need not be analyzed multiple times. Therefore, the derivation rules are parametric in a function context Δ that we leave implicit in the rules presented here. More details can be found in the extended version.

Derivation Rules. The rules of our derivation system must serve two purposes. They must *attach* potential to certain program variable intervals and use this potential, when it is allowed, to *pay* for resource consuming operations. These two purposes are illustrated on the Q:SKIP rule. This rule reuses its precondition as postcondition, it is explained by two facts: First, no resource is consumed by the skip operation, thus no potential has to be used to pay for the evaluation. Second, the program state is not changed by the execution of a skip statement. Thus all potential available before the execution of the skip statement is still available after.

The rules Q:INCP, Q:DECP, and Q:INC describe how the potential is distributed after a size change of a variable. The rule Q:INCP is for increments $x \leftarrow x + y$ and Q:DECP is for decrements $x \leftarrow x - y$. They both apply only when we can deduce from the logical context Γ that $y \geqslant 0$. Of course, there are symmetrical rules Q:INCN and Q:DECN (not presented here) that can be applied if y is negative. The rules are all equivalent in the case where $y = 0$. The rule Q:INC can be applied if we cannot find the sign of y.

To explain how rules for increment and decrement work, it is sufficient to understand the rule Q:INCP. The others follow the same idea and are symmetrical. In Q:INCP, the program updates a variable x with $x + y$ where $y \geqslant 0$. Since x is changed, the quantitative annotation must be updated to reflect the change of the program state. We write x' for the value of x after the assignment. Since x is the only variable changed, only intervals of the form $[u, x]$ and $[x, u]$ will be resized. Note that for any u, $[x, u]$ will get smaller with the update, and if $x' \in [x, u]$ we have $|[x, u]| = |[x, x']| + |[x', u]|$. But $|[x, x']| = |[0, y]|$ which means that the potential q'_{0y} in the postcondition can be increased by q_{xu} under the guard that $x' \in [x, u]$. Dually, the interval $[v, x]$ can get bigger with the update. We know that $|[v, x']| \leqslant y + |[v, x]|$. So we decrease the potential of $[0, y]$ by q_{vx} to pay for this change. The rule ensures this only for $v \notin \mathcal{U}$ because $x \leqslant v$ otherwise, and thus $|[v, x]| = 0$.

The rule Q:LOOP is a cornerstone of our analysis. To apply it on a loop body, one needs to find an invariant potential Q that will pay for the iterations. At each iteration, M_l resources are spent to jump back. This explains the postcondition $Q + M_l$. Since the loop can only be exited with a break statement, the postcondition $\{\Gamma'; Q'\}$ for

$$\frac{}{B, R \vdash \{\Gamma; Q\}\ \mathsf{skip}\ \{\Gamma; Q\}}\,(\text{Q:SKIP}) \qquad \frac{}{B, R \vdash \{\Gamma; Q{+}M_a\}\ \mathsf{assert}\ e\ \{\Gamma \wedge e; Q\}}\,(\text{Q:ASSERT})$$

$$\frac{}{B, R \vdash \{\Gamma; Q{+}M_t(n)\}\ \mathsf{tick}(n)\ \{\Gamma; Q\}}\,(\text{Q:TICK}) \qquad \frac{}{(\Gamma; Q_B), R \vdash \{\Gamma; Q_B{+}M_b\}\ \mathsf{break}\ \{\Gamma'; Q'\}}\,(\text{Q:BREAK})$$

$$\frac{P = Q_R[ret/x] \qquad \Gamma = \Gamma_R[ret/x] \qquad \forall i \in \mathrm{dom}(P).\ p_i = q_i}{B, (\Gamma_R; Q_R) \vdash \{\Gamma; Q\}\ \mathsf{return}\ x\ \{\Gamma'; Q'\}}\,(\text{Q:RETURN}) \qquad \frac{(\Gamma'; Q'), R \vdash \{\Gamma; Q\}\ S\ \{\Gamma; Q{+}M_l\}}{B, R \vdash \{\Gamma; Q\}\ \mathsf{loop}\ S\ \{\Gamma'; Q'\}}\,(\text{Q:LOOP})$$

$$\frac{B, R \vdash \{\Gamma; Q\}\ S_1\ \{\Gamma'; Q'{+}M_s\} \qquad B, R \vdash \{\Gamma'; Q'\}\ S_2\ \{\Gamma''; Q''\}}{B, R \vdash \{\Gamma; Q\}\ S_1; S_2\ \{\Gamma''; Q''\}}\,(\text{Q:SEQ}) \qquad \frac{\begin{array}{c}B, R \vdash \{\Gamma \wedge e; Q{-}M_c^1)\}\ S_1\ \{\Gamma'; Q'\} \\ B, R \vdash \{\Gamma \wedge \neg e; Q{-}M_c^2\}\ S_2\ \{\Gamma'; Q'\}\end{array}}{B, R \vdash \{\Gamma; Q{+}M_e(e)\}\ \mathsf{if}(e)\ S_1\ \mathsf{else}\ S_2\ \{\Gamma'; Q'\}}\,(\text{Q:IF})$$

$$\frac{\begin{array}{c}\Gamma \models y \geqslant 0 \qquad \mathcal{U} = \{u \mid \Gamma \models x + y \in [x, u]\} \\ q'_{0y} = q_{0y} + \sum_{u \in \mathcal{U}} q_{xu} - \sum_{v \notin \mathcal{U}} q_{vx}\end{array}}{B, R \vdash \{\Gamma[x/x{+}y]; Q{+}M_u{+}M_e(x{+}y)\}\ x \leftarrow x + y\ \{\Gamma; Q'\}}\,(\text{Q:INCP}) \qquad \frac{\begin{array}{c}M = M_u + M_e(x \pm y) \\ q'_{0y} = q_{0y} - \sum_v q_{vx} \qquad q'_{y0} = q_{y0} - \sum_v q_{xv}\end{array}}{B, R \vdash \{\Gamma[x/x \pm y]; Q{+}M\}\ x \leftarrow x \pm y\ \{\Gamma; Q'\}}\,(\text{Q:INC})$$

$$\frac{\begin{array}{c}q'_{xy}, q'_{yx} \in \mathbb{Q}_0^+ \\ \forall u.(q_{yu} = q'_{xu} + q'_{yu} \wedge q_{uy} = q'_{ux} + q'_{uy})\end{array}}{B, R \vdash \{\Gamma[x/y]; Q{+}M_u{+}M_e(y)\}\ x \leftarrow y\ \{\Gamma; Q'\}}\,(\text{Q:SET}) \qquad \frac{\begin{array}{c}\Gamma \models y \geqslant 0 \qquad \mathcal{U} = \{u \mid \Gamma \models x - y \in [u, x]\} \\ q'_{y0} = q_{y0} + \sum_{u \in \mathcal{U}} q_{ux} - \sum_{v \notin \mathcal{U}} q_{xv}\end{array}}{B, R \vdash \{\Gamma[x/x{-}y]; Q{+}M_u{+}M_e(x{-}y)\}\ x \leftarrow x - y\ \{\Gamma; Q'\}}\,(\text{Q:DECP})$$

$$\frac{\begin{array}{c}(\Gamma_f; Q_f, \Gamma'_f; Q'_f) \in \Delta(f) \qquad \mathrm{Loc} = \mathrm{Locals}(Q) \qquad \forall i \neq j.\ x_i \neq x_j \qquad c \in \mathbb{Q}_0^+ \qquad Q = P + S \qquad Q' = P' + S \qquad U = Q_f[a\vec{r}gs/\vec{x}] \\ U' = Q'_f[ret/r] \qquad \forall i \in \mathrm{dom}(U).\ p_i = u_i \qquad \forall i \in \mathrm{dom}(U').\ p'_i = u'_i \qquad \forall i \notin \mathrm{dom}(U').\ p'_i = 0 \qquad \forall i \notin \mathrm{Loc}.\ s_i = 0\end{array}}{B, R \vdash \{\Gamma_f[a\vec{r}gs/\vec{x}] \wedge \Gamma_{\mathrm{Loc}}; Q{+}c{+}M_f\}\ r \leftarrow f(\vec{x})\ \{\Gamma'_f[ret/r] \wedge \Gamma_{\mathrm{Loc}}; Q'{+}c{-}M_r\}}\,(\text{Q:CALL})$$

$$\frac{\begin{array}{c}\Sigma f = (\vec{y}, S_f) \\ B, (\Gamma'_f; Q'_f) \vdash \{\Gamma_f[a\vec{r}gs/\vec{y}]; Q_f[a\vec{r}gs/\vec{y}]\}\ S_f\ \{\Gamma'; Q'\}\end{array}}{(\Gamma_f; Q_f, \Gamma'_f; Q'_f) \in \Delta(f)}\,(\text{Q:EXTEND}) \qquad \frac{\begin{array}{c}B, R \vdash \{\Gamma_2; Q_2\}\ S\ \{\Gamma'_2; Q'_2\} \qquad \Gamma_1 \models \Gamma_2 \\ Q_1 \geq_{\Gamma_1} Q_2 \qquad \Gamma'_2 \models \Gamma'_1 \qquad Q'_2 \geq_{\Gamma'_2} Q'_1\end{array}}{B, R \vdash \{\Gamma_1; Q_1\}\ S\ \{\Gamma'_1; Q'_1\}}\,(\text{Q:WEAK})$$

$$\frac{\begin{array}{c}\mathcal{L} = \{xy \mid \exists l_{xy} \in \mathbb{N}.\ \Gamma \models l_{xy} \leqslant |[x, y]|\} \qquad \mathcal{U} = \{xy \mid \exists u_{xy} \in \mathbb{N}.\ \Gamma \models |[x, y]| \leqslant u_{xy}\} \\ \forall i \in \mathcal{U}.\ q'_i \geqslant q_i - r_i \qquad \forall i \in \mathcal{L}.\ q'_i \geqslant q_i + p_i \qquad \forall i \notin \mathcal{U} \cup \mathcal{L} \cup \{0\}.\ q'_i \geqslant q_i \qquad q'_0 \geqslant q_0 + \sum_{i \in \mathcal{U}} u_i r_i - \sum_{i \in \mathcal{L}} l_i p_i\end{array}}{Q' \geq_\Gamma Q}\,(\text{RELAX})$$

Figure 4. Inference rules of the quantitative analysis.

the statement loop S is used as break postcondition in the derivation for S.

Another interesting rule is Q:CALL. It needs to account for the changes to the stack caused by the function call, the arguments/return value passing, and the preservation of local variables. We can sum up the main ideas of the rule as follows.

- The potential in the pre- and postcondition of the function specification is equalized to its matching potential in the callee's pre- and postcondition.
- The potential of intervals $|[x, y]|$ is preserved across a function call if x and y are local.
- The unknown potentials after the call (e.g. $|[x, g]|$, with x local and g global) are set to zero in the postcondition.

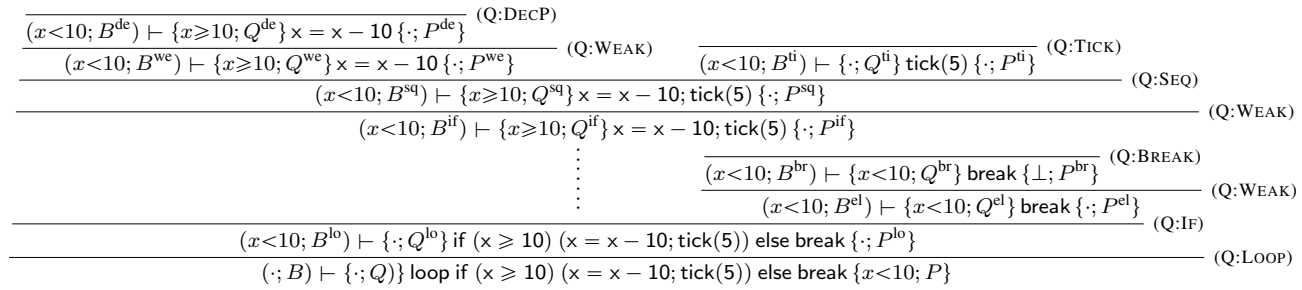
If x and y are local variables and $f(x, y)$ is called, Q:CALL splits the potential of $|[x, y]|$ in two parts. One part to perform the computation in the function f and one part to keep for later use after the function call. This splitting is realized by the equations $Q = P{+}S$ and $Q' = P'{+}S'$. Arguments in the function precondition $(\Gamma_f; Q_f)$ are named using a fixed vector $a\vec{r}gs$ of names different from all program variables. This prevents name conflicts and ensures that the substitution $[a\vec{r}gs/\vec{x}]$ is meaningful. Symmetrically, we use the unique name ret to represent the return value in the function's postcondition $(\Gamma'_f; Q'_f)$.

The rule Q:WEAK is the only rule that is not syntax directed. We could integrate weakenings into every syntax directed rule but, for the sake of efficiency, the implementation uses a simple heuristic instead. The high-level idea of Q:WEAK is the following: If we have a sound judgement, then it is sound to add more potential to the precondition and remove potential from the postcondition. The concept of *more potential* is formalized by the relation $Q' \geq_\Gamma Q$ that is defined in the rule RELAX. This rule also deals with the important task of transferring constant potential (represented by q_0) to interval sizes and vice versa. If we can deduce from the logical context that the interval size $|[x, y]| \geqslant \ell$ is larger than a constant ℓ then we can turn the potential $q_{xy} \cdot |[x, y]|$ form the interval into the constant potential $\ell \cdot q_{xy}$ and guarantee that we do not gain potential. Conversely, if $|[x, y]| \leqslant u$ for a constant u then we can transfer constant potential $u \cdot q_{xy}$ to the interval potential $q_{xy} \cdot |[x, y]|$ without gaining potential.

5. Automatic Inference via LP Solving

We separate the search of a derivation in two steps. As a first step we go through the functions of the program and apply inductively the derivation rules of the automatic amortized analysis. This is done in a bottom-up way for each strongly connected component (SCC) of the call graph. During this process our tool uses symbolic names for the rational coefficients q_i in the rules. Each time a linear constraint must be satisfied by these coefficients, it is recorded in a global list for the SCC using the symbolic names. We reuse the constraint list for every call from outside the SCC.

We then feed the collected constraints to an off-the-shelf LP solver (currently CLP [19]). If the solver successfully finds a solution, we know that a derivation exists and extract the values for the initial Q from the solver to get a resource bound for the program. To get a full derivation, we extract the complete solution from the



Constraints:

$P{=}B^{\text{lo}} \wedge Q{=}Q^{\text{lo}}{=}P^{\text{lo}}$ $\quad B^{\text{el}}{=}B^{\text{if}}{=}B^{\text{lo}} \wedge Q^{\text{el}}{=}Q^{\text{if}}{=}Q^{\text{lo}} \wedge P^{\text{el}}{=}P^{\text{if}}{=}P^{\text{lo}}$ $\quad B^{\text{el}}{=}B^{\text{br}} \wedge Q^{\text{el}} \geq_{(x<10)} Q^{\text{br}} \wedge P^{\text{br}} \geq_{(\cdot)} P^{\text{el}}$

$B^{\text{br}}{=}Q^{\text{br}}$ $\quad B^{\text{if}}{=}B^{\text{sq}} \wedge Q^{\text{if}} \geq_{(x<10)} Q^{\text{sq}} \wedge P^{\text{sq}} \geq_{(\cdot)} P^{\text{if}}$ $\quad B^{\text{sq}}{=}B^{\text{we}}{=}B^{\text{ti}} \wedge Q^{\text{sq}}{=}Q^{\text{we}} \wedge P^{\text{we}}{=}Q^{\text{ti}} \wedge P^{\text{ti}}{=}P^{\text{sq}}$

$Q^{\text{ti}}{=}P^{\text{ti}} + 5$ $\quad B^{\text{we}}{=}B^{\text{de}} \wedge Q^{\text{we}} \geq_{(x<10)} Q^{\text{de}} \wedge P^{\text{de}} \geq_{(\cdot)} P^{\text{we}}$ $\quad p^{\text{de}}_{0,10}{=}q^{\text{de}}_{0,10} + q^{\text{de}}_{0,x} \wedge p^{\text{de}}_{0}{=}q^{\text{de}}_{0} \wedge \forall(\alpha,\beta) \neq (0,10).\, p^{\text{de}}_{\alpha,\beta}{=}q^{\text{de}}_{\alpha,\beta}$

Linear Objective Function: $1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x}$ Constant Objective Function: $1{\cdot}q_0 + 11{\cdot}q_{0,10}$

Figure 5. An example derivation as produced C^4B. The constraints are resolved by an off-the-shelf LP solver.

solver and apply it to the symbolic names q_i of the coefficients in the derivation. If the LP solver fails to find a solution, an error is reported.

Figure 5 contains an example derivation as produced by C^4B. The upper case letters (with optional superscript) such as Q^{de} are families of variables that are later part of the constraint system that is passed to the LP solver. For example Q^{de} stands for the potential function $q^{\text{de}}_0 + q^{\text{de}}_{x,0}|[x,0]| + q^{\text{de}}_{0,x}|[0,x]| + q^{\text{de}}_{x,10}|[x,10]| + q^{\text{de}}_{10,x}|[10,x]| + q^{\text{de}}_{0,10}|[0,10]|$, where the variables such as $q^{\text{de}}_{x,10}$ are yet unknown and later instantiated by the LP solver.

In general, the weakening rule can be applied after every syntax directed rule. However, it can be left out in practice at some places to increase the efficiency of the tool. The weakening operation $\geq_\Gamma$ is defined by the rule RELAX. It is parameterized by a logical context that is used to gather information on interval sizes. For example,

$$\begin{aligned} P^{\text{de}} \geq_{(\cdot)} P^{\text{we}} \equiv\ & p^{\text{we}}_{0,10} \leqslant p^{\text{de}}_{0,10} + u_{0,10} - v_{0.10} \\ & \wedge\ p^{\text{we}}_{0} \leqslant p^{\text{de}}_{0} - 10{\cdot}u_{0,10} + 10{\cdot}v_{0.10} \\ & \wedge\ \forall(\alpha,\beta) \neq (0,10).\, p^{\text{we}}_{\alpha,\beta} \leqslant p^{\text{de}}_{\alpha,\beta}\ . \end{aligned}$$

The other rules are syntax directed and applied inductively. For example, the outermost expression is a loop, so we use the rule Q:Loop at the root of the derivation tree. At this point, we do not know yet whether a loop invariant exists. But we produce the constraints $Q^{\text{lo}} = P^{\text{lo}}$. These constraints express the fact that the potential functions before and after the loop body are equal and thus constitute an invariant.

After the constraint generation, the LP solver is provided with an objective function to be minimized. We wish to minimize the initial potential, which is a resource bound on the whole program. Here it is given by Q. Moreover, we would like to express that minimization of linear potential such as $q_{10,x}|[10,x]|$ takes priority over minimization of constant potential such as $q_{0,10}|[0,10]|$.

To get a tight bound, we use modern LP solvers that allow constraint solving and minimization at the same time: First we consider our initial constraint set as given in Figure 5 and ask the solver to find a solution that satisfies the constraints and minimizes the linear expression $1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x}$. The penalties given to certain factors are used to prioritize certain intervals. For example, a bound with $[10,x]$ will be preferred to another with $[0,x]$ because $|[10,x]| \leqslant |[0,x]|$. The LP solver now returns a solution of the constraint set and an objective value. The solver also memorizes the optimization path that led to the optimal solution. In this case, the objective value would be 5000 since the LP solver assigns $q_{0,x} = 0.5$ and $q_* = 0$ otherwise. We now add the constraint $1{\cdot}q_{x,0} + 10000{\cdot}q_{0,x} + 11{\cdot}q_{x,10} + 9990{\cdot}q_{10,x} \leqslant 5000$ to our constraint set and ask the solver to optimize the objective function $q_0 + 11{\cdot}q_{0,10}$. This happens in almost no time in practice. The final solution is $q_{0,x} = 0.5$ and $q_* = 0$ otherwise. Thus the derived bound is $0.5|[0,x]|$.

A notable advantage of the LP-based approach compared to SMT-solver–based techniques is that a satisfying assignment is a proof certificate instead of a counter example. To provide high-assurance bounds, this certificate can be checked in linear time by a simple validator.

6. Logical State and User Interaction

While complete automation is desirable, it is not always possible since the problem of bound derivation is undecidable. In this section we present a new technique to derive complex resource bounds semi-automatically by leveraging our automation. Our goal is to develop an interface between bound derivation and established qualitative verification techniques.

When the resource bound of a program depends on the contents of the heap, or is non-linear (e.g. logarithmic, exponential), we introduce a *logical state* using *auxiliary variables*. Auxiliary variables guide C^4B during bound derivation but they do not change the behavior of the program.

More precisely, the technique consists of the following steps. First, a program P that fails to be analyzed automatically is enriched by auxiliary variables $\vec{x}$ and assertions to form a program $P_l(\vec{x})$. Second, an initial value $\vec{X}(\sigma)$ for the logical variables is selected to satisfy the proposition:

$$\forall n\, \sigma\, \sigma'.\, (\sigma, P_l(\vec{X}(\sigma))) \Downarrow_n \sigma' \Longrightarrow \exists n' \leqslant n.\, (\sigma, P) \Downarrow_{n'} \sigma'. \quad (*)$$

Since the annotated program and the original one are usually syntactically close, the proof of this result goes by simple induction on the resource-aware evaluation judgement. Third, using existing automation tools, a bound $B(\vec{x})$ for $P_l(\vec{x})$ is derived. Finally this bound, instantiated with $\vec{X}$, gives the final resource bound for the program P.

This idea is illustrated by the program in Figure 6. The parts of the code in blue are annotations that were added to the original program text. The top-level loop increments a binary counter k times. A naive analysis of the algorithm yields the quadratic bound $k \cdot N$. However, the algorithm is in fact linear and its cost is bounded

```
1  logical state invariant {na = #1(a)}
2  while (k > 0) {
3    x=0;
4    while (x < N && a[x] == 1) {
5      assert(na > 0);
6      a[x]=0; na--;
7      tick(1); x++; }
8    if (x < N) { a[x]=1; na++; tick(1); }
9    k--;
10 }
```

Figure 6. Assisted bound derivation using logical state. We write $\#_1(a)$ for $\#\{i \mid 0 \leqslant i < N \wedge a[i]=1\}$ and use the tick metric. The derived bound is $2|[0,k]| + |[0,na]|$.

```
1  logical state invariant {lg > log2(h − l)}
2  bsearch(x,l,h,lg) {
3    if (h-l > 1) {
4      assert(lg > 0);
5      m = l + (h-l)/2;
6      lg--; if (a[m]>x) h=m; else l=m;
7      tick(M_bsearch);
8      l = bsearch(x,l,h,lg);
9      tick(−M_bsearch);
10   } else return l;
11 }
```

Figure 7. Assisted bound derivation using logical state. We write $\log_2(x)$ for the integer part of logarithm of x in base 2. The semi-automatically derived bound is $|[0, lg]|$.

by $2k + \#_1(a)$ where $\#_1(a)$ denotes the number of one entries in the array a. Since this number depends on the heap contents, no tool available for C is able to derive the linear bound. However, it can be inferred by our automated tool if a logical variable na is introduced. This logical variable is a reification of the number $\#_1(a)$ in the program. For example, on line 6 of the example we are setting `a[x]` to 0 and because of the condition we know that this array entry was 1. To reflect this change on $\#_1(a)$, the logical variable na is decremented. Similarly, on line 8, an array entry which was 0 becomes 1, so na is incremented. To complete the step 2 of the systematic procedure described above, we must show that the extra assertion `na > 0` on line 5 cannot fail. We do it by proving inductively that $na = \#_1(a)$ and remarking that since `a[x] == 1` is true, we must have $\#_1(a) > 0$, thus the assertion `na > 0` never fails.

Another simple example is given in Figure 7 where a logarithmic bound on the stack consumption of a binary search program is proved using logical variable annotations. Once again, annotations are in blue in the program text. In this example, to ease the proof of equivalence between the annotated program and the original one, we use the inequality $lg > \log_2(h - l)$ as invariant. This allows a simpler proof because, when working with integer arithmetic, it is not always the case that $\log_2(x - x/2) = \log_2(x) - 1$.

Generally, we observed that because the instrumented program is structurally same as the original one, it is enough to prove that the added assertions never fail in order to show the two programs satisfy the proposition (*). This can usually be piggybacked on standard static-analysis tools.

7. Soundness Proof

The soundness of the analysis builds on a new cost semantics for Clight and an extended quantitative logic. Using these two tools, the soundness of the automatic analysis described in Section 3 is proved by a translation morphism to the logic.

The main parts of the soundness proof are formalized with Coq and available for download. The full definitions of the cost semantics and the quantitative Hoare logic, and more details on the soundness proof can be found in the extended version of this article.

Cost Semantics for Clight. To base the soundness proof on a formal ground, we start by defining a new cost-aware operational semantics for Clight. Clight's operational semantics is based on small-step transitions and continuations. Expressions—which do not have side effects—are evaluated in a big-step fashion.

A program state $\sigma = (\theta, \gamma)$ is composed of two maps from variable names to integers. The first map, θ : Locals $\to \mathbb{Z}$, assigns integers to local variables of a function, and the second map, γ : Globals $\to \mathbb{Z}$, gives values to global variables of the program. In this article, we assume that all values are integers but in the implementation we support all data types of Clight. The evaluation function $[\![\cdot]\!]$ maps an expression $e \in E$ to a value $[\![e]\!]_\sigma \in \mathbb{Z}$ in the program state σ. We write $\sigma(x)$ to obtain the value of x in program state σ. Similarly, we write $\sigma[x \mapsto v]$ for the state based on σ where the value of x is updated to v.

The small-step semantics is standard, except that it tracks the resource consumption of a program. The semantics is parametric in the resource of interest for the user of our system. We achieve this independence by parameterizing evaluations with a resource *metric* M; a tuple of rational numbers and two maps. Each of these parameters indicates the amount of resource consumed by a corresponding step in the semantics. Resources can be released by using a negative cost. Two sample rules for update and tick follow.

$$\frac{\sigma' = \sigma[x \mapsto [\![e]\!]_\sigma]}{(\sigma, x \leftarrow e, K, c) \to (\sigma', \mathsf{skip}, K, c - M_u - M_e(e))}\ (\mathrm{U}) \qquad \frac{}{(\sigma, \mathsf{tick}(n), K, c) \to (\sigma, \mathsf{skip}, K, c - M_t(n))}\ (\mathrm{T})$$

The rules have as implicit side condition that c is non-negative. This makes it possible to detect a resource crash as a stuck configuration where $c < 0$.

Quantitative Hoare Logic. To prove the soundness of C^4B we found it useful to go through an intermediate step using a quantitative Hoare logic. This logic is at the same time a convenient semantic tool and a clean way to interface manual proofs with our automation. We base it on a logic for stack usage [15], add support for arbitrary resources, and simplify the handling of auxiliary state.

We define quantitative Hoare triples as $B; R \vdash_L \{Q\}\, S\, \{Q'\}$ where B, R, Q, and Q' are maps from program states to an element of $\mathbb{Q}_0^+ \cup \{\infty\}$ that represents an amount of resources available. The assertions B and R are postconditions for the case in which the block S exits by a `break` or `return` statement. Additionally, R depends on the return value of the current function. The meaning of the triple $\{Q\}\, S\, \{Q'\}$ is as follows: If S is executed with starting state σ, the empty continuation, and at least $Q(\sigma)$ resource units available then the evaluation does not run out of resources and there are at least $Q'(\sigma')$ resources left if the evaluation terminates in σ'. The logic rules are similar to the ones in previous work and generalized to account for the cost introduced by our cost-aware semantics.

Finally, we define a strong compositional continuation-based soundness for triples and prove the validity of all the rules in Coq. The full version of this paper [16], provides explanations for the rules and a thorough overview of our soundness proof.

The Soundness Theorem. We use the quantitative logic as the target of a translation function for the automatic derivation system. This reveals two orthogonal aspects of the proof: on one side, it relies on amortized reasoning (the quantitative logic rules), and on the other side, it uses combinatorial properties of our linear potential functions (the automatic analysis rules).

Technically, we define a translation function $\mathcal{T}$ such that if a judgement J in the automatic analysis is derivable, $\mathcal{T}(J)$ is deriv-

	t09	t19	t30	t15	t13

	`i=1; j=0;`
	`while (j<x) {`
	`  j++;`
	`  if (i>=4)`
	`  i=1, tick(40);`
	`  else i++;`
	`  tick(1); }`
		`while (i>100) {`
		`  i--; tick(1);`
		`} i += k+50;`
		`while (i>=0) {`
		`  i--; tick(1);`
		`}`
			`while (x>0) {`
			`  x--;`
			`  t=x, x=y, y=t;`
			`  tick(1);`
			`}`
				`assert(y>=0);`
				`while (x > y) {`
				`  x -= y+1;`
				`  for (z=y; z>0; z--)`
				`  tick(1);`
				`  tick(1);`
				`}`
					`while (x>0) {`
					`  x--;`
					`  if (*) y++;`
					`  else`
					`  while (y>0)`
					`  y--, tick(1);`
					`  tick(1); }`

C^4B	$11\lvert[0,x]\rvert$	$50+\lvert[-1,i]\rvert+\lvert[0,k]\rvert$	$\lvert[0,x]\rvert+\lvert[0,y]\rvert$	$\lvert[0,x]\rvert$	$2\lvert[0,x]\rvert+\lvert[0,y]\rvert$
Rank	$23{\cdot}x-14$	$54+k+i$	—	$2+2x-y$	$0.5{\cdot}y^2{+}yx\ldots$
LOOPUS	$41\max(x,0)$	$\max(i{-}100,0)$ $+\max(k{+}i{+}51,0)$	—	—	$2\max(x,0)$ $+\max(y,0)$

Figure 8. Comparison of resource bounds derived by different tools on several examples with linear bounds.

able in the quantitative logic. By using $\mathcal{T}$ to translate derivations of the automatic analysis to derivations in the quantitative logic we can directly obtain a certified resource bound for the analyzed program.

The translation of an assertion $(\Gamma; Q)$ in the automatic analysis is defined by

$$\mathcal{T}(\Gamma; Q) := \lambda\sigma.\, \Gamma(\sigma) + \Phi_Q(\sigma),$$

where we write Φ_Q for the unique linear potential function defined by the quantitative annotation Q. The logical context Γ is implicitly lifted to a quantitative assertion by mapping a state σ to 0 if $\Gamma(\sigma)$ holds and to ∞ otherwise. These definitions let us translate the judgement $J := B, R \vdash \{P\}\, S\, \{P'\}$ by

$$\mathcal{T}(J) := \mathcal{T}(B); \mathcal{T}(R) \vdash_L \{\mathcal{T}(P)\}\, S\, \{\mathcal{T}(P')\}.$$

The soundness of the automatic analysis can now be stated formally with the following theorem.

Theorem 1 (Soundness of the automatic analysis)**.** *If J is a judgement derived by the automatic analysis, then $\mathcal{T}(J)$ is a quantitative Hoare triple derivable in the quantitative logic.*

The proof of this theorem is constructive and maps each rule of the automatic analysis directly to its counterpart in the quantitative logic. The trickiest parts are the translations of the rules for increments and decrements and the rule Q:WEAK for weakening because they make essential use of the algebraic properties of the potential functions.

8. Experimental Evaluation

We have experimentally evaluated the practicality of our automatic amortized analysis with more than 30 challenging loop and recursion patterns from open-source code and the literature [20–22]. A full list of examples is given in the extended version [16].

Figure 8 shows five representative loop patterns from the evaluation. Example *t09* is a loop that performs an expensive operation every 4 steps. C^4B is the only tool able to amortize this cost over the input parameter x. Example *t19* demonstrates the compositionality of the analysis. The program consists of two loops that decrement a variable i. In the first loop, i is decremented down to 100 and in the second loop i is decremented further down to -1. However, between the loops we assign `i += k+50`. So in total the program performs $52 + |[-1, i]| + |[0, k]|$ ticks. Our analysis finds this tight bound because our amortized analysis naturally takes into account the relation between the two loops. Example *t30* decrements both input variables x and y down to zero in an unconventional way. In the loop body, first x is decremented by one, then the values of the variables x and y are switched using the local variable t as a buffer. Our analysis infers the tight bound $|[0, x]| + |[0, y]|$. Sometimes we need some assumptions on the inputs in order to derive a bound. Example *t15* is such a case. We assume here that the input variable y is non-negative and write `assert(y>=0)`. The assignment `x -= y+1` in the loop is split in `x--` and `x -= y`. If we enter the loop then we know that $x > 0$, so we can obtain constant potential from `x--`. Then we know that $x \geqslant y \geqslant 0$, as a consequence we can share the potential of $|[0, x]|$ between $|[0, x]|$ and $|[0, y]|$ after `x -= y`.

Example *t13* shows how amortization can be used to find linear bounds for nested loops. The outer loop is iterated $|[0, x]|$ times. In the conditional, we either (the branching condition is arbitrary) increment the variable y or we execute an inner loop in which y is counted back to 0. C^4B computes a tight bound. The extended version also contains a discussion of the automatic bound derivation for the Knuth-Morris-Pratt algorithm for string search. C^4B finds the tight linear bound $1 + 2|[0, n]|$.

Table 1. Comparison of C^4B with other automatic tools.

	KoAT	Rank	LOOPUS	SPEED	C^4B
#bounds	9	24	20	14	32
#lin. bounds	9	21	20	14	32
#best bounds	0	0	11	14	29
#tested	14	33	33	14	33

To compare our tool with existing work, we focused on loop bounds and use a simple metric that counts the number of back edges (i.e., number of loop iterations) that are followed in the execution of the program because most other tools only bound this specific cost. In Figure 8, we show the bounds we derived (C^4B) together with the bounds derived by LOOPUS [38] and Rank [3]. We also contacted the authors of SPEED but have not been able to obtain this tool. KoAT [13] and PUBS [1] currently cannot operate on C code and the examples would need to be manually translated into a term-rewriting system to be analyzed by these tools. For Rank it is not completely clear how the computed bound relates to the C program since the computed bound is for transitions in an automaton that is derived from the C code. For instance, the bound $2 + y - x$ that is derived for *t08* only applies to the first loop in the program.

Table 1 summarizes the results of our experiments presented in Appendix A. It shows for each tool the number of derived bounds (#bounds), the number of asymptotically tight bounds (#lin. bounds), the number of bounds with the best constant factors in comparison with the other tools (#best bounds), and the number of examples that we were able to test with the tool (#tested). Since we were not able to run the experiments for KoAT and SPEED, we simply used the bounds that have been reported by the authors of the respective tools. The results show that our automatic amortized analysis outperforms the existing tools on our example programs. However, this experimental evaluation has to be taken with a grain of salt. Existing tools complement C^4B since they can derive polynomial bounds and support more features of C. We were particularly impressed by LOOPUS which is very robust, works on large C files, and derives very precise bounds.

Table 2 contains a compilation of the results of our experiments with the cBench benchmark suite. It shows a representative list of automatically derived function bounds. In total we analyzed more

Table 2. Derived bounds for functions from cBench.

Function	LoC	Bound	Time (s)
adpcm_coder	145	$1 + \|[0, \mathsf{N}]\|$	0.6
adpcm_decod	130	$1 + \|[0, \mathsf{N}]\|$	0.2
BF_cfb64_enc	151	$1 + 2\|[-1, \mathsf{N}]\|$	0.7
BF_cbc_enc	180	$2 + 0.25\|[-8, \mathsf{N}]\|$	1.0
mad_bit_crc	145	$61.19{+}0.19\|[-1, \mathsf{N}]\|$	0.4
mad_bit_read	65	$1 + 0.12\|[0, \mathsf{N}]\|$	0.05
MD5Update	200	$133.95{+}1.05\|[0, \mathsf{N}]\|$	1.0
MD5Final	195	141	0.22
sha_update	98	$2 + 3.55\|[0, \mathsf{N}]\|$	1.2
PackBitsDecode	61	$1 + 65\|[-129, \mathsf{cc}]\|$	0.6
KMPSearch	20	$1 + 2\|[0, n]\|$	0.1
ycc_rgb_conv	66	$\mathsf{nr} \cdot \mathsf{nc}$	0.1
uv_decode	31	$\log_2(\mathsf{UV\_NVS}) + 1$	0.1

than 2900 lines of code. In the LoC column we not only count the lines of the analyzed function but also the ones of all the function it calls. We analyzed the functions using a metric that assigns a cost 1 to all the back-edges in the control flow (loops, and function calls). The bounds for the functions ycc_rgb_conv and uv_decode have been inferred with user interaction as described in Section 6. The most challenging functions for C^4B have unrolled loops where many variables are assigned. This stresses our analysis because the number of LP variables has a quadratic growth in program variables. Even on these stressful examples, the analysis could finish in less than 2 seconds. For example, the sha_update function is composed of one loop calling two helper functions that in turn have 6 and 1 inner loops. In the analysis of the SHA algorithm, the compositionality of our analysis is essential to get a tight bound since loops on the same index are sequenced 4 and 2 times without resetting it. All other tools derive much larger constant factors.

With our formal cost semantics, we can run our examples for different inputs and measure the cost to compare it to our derived bound. Figure 9 shows such a comparison for Example *t08*, a variant of *t08a* from Section 3. One can see that the derived constant factors are the best possible if the input variable x is non-negative.

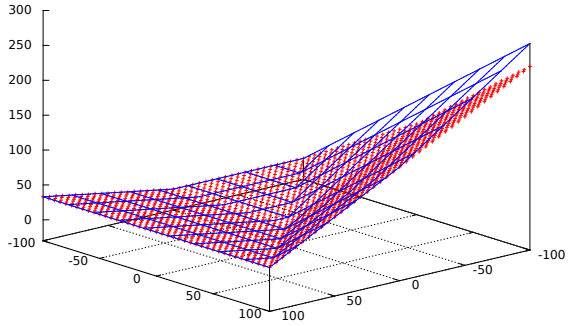


Figure 9. The automatically derived bound $1.33|[x, y]| + 0.33|[0, x]|$ (blue lines) and the measured runtime cost (red crosses) for Example *t08*. For $x \geqslant 0$ the bound is tight.

9. Limitations

Our implementation does not currently support all of Clight. Programs with function pointers, goto statements, continue statements, and pointers to stack-allocated variables cannot be analyzed automatically. While these limitations concern the current implementation, our technique is in principle capable to handle them.

For the sake of simplicity, the automated system described here is restricted to finding only linear bounds. However, the amortized analysis technique was shown to work with polynomial bounds [25]; we leave this extension of our system as future work.

Even certain linear programs cannot be analyzed automatically by C^4B, it is usually the case for programs that rely on heap invariants (like nul-terminated C strings), for programs in which resource usage depends on the result of non-linear operations (like $\%$ or $*$) in a non-trivial way, or for programs whose termination can only be proved by complex path-sensitive reasoning.

10. Related Work

Our work has been inspired by type-based amortized resource analysis for functional programs [23, 26, 28]. Here, we present the first automatic amortized resource analysis for C. None of the existing techniques can handle the example programs we describe in this work. The automatic analysis of realistic C programs is enabled by two major improvements over previous work. First, we extended the analysis system to associate potential with not just individual program variables but also multivariate intervals and, more generally, auxiliary variables. In this way, we solved the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on (possibly negative) integers without decreasing one individual number in each iteration. Second, for the first time, we have combined an automatic amortized analysis with a system for interactively deriving bounds. In particular, recent systems [24] that deal with integers and arrays cannot derive bounds that depend on values in mutable locations, possibly negative integers, or on differences between integers.

A recent project [15] has implemented and verified a quantitative logic to reason about stack-space usage, and modified the verified CompCert C compiler to translate C level bound to x86 stack bounds. This quantitative logic is also based on the potential method but has very rudimentary support for automation. It is not based on efficient LP solving and cannot automatically derive symbolic bounds. In contrast, our main contribution is an automatic amortized analysis for C that can derive parametric bounds for loops and recursive functions fully automatically. We use a more general quantitative Hoare logic that is parametric over the resource of interest.

There exist many tools that can automatically derive loop and recursion bounds for imperative programs such as SPEED [20, 22], KoAT [13], PUBS [1], Rank [3], ABC [10] and LOOPUS [38, 40]. These tools are based on abstract interpretation–based invariant generation and/or term rewriting techniques, and they derive impressive results on realistic software. The importance of amortization to derive tight bounds is well known in the resource analysis community [4, 30, 38]. Currently, the only other available tools that can be directly applied to C code are Rank and LOOPUS. As demonstrated, C^4B is more compositional than the aforementioned tools. Our technique, is the only one that can generate resource specifications for functions, deal with resources like memory that might become available, generate proof certificates for the bounds, and support user guidance that separates qualitative and quantitative reasoning.

There are techniques [12] that can compute the memory requirements of object oriented programs with region-based garbage collection. These systems can handle loops but not recursive or composed functions. We are only aware of two verified quantitative analysis systems. Albert et al. [2] rely on the KeY tool to automatically verify previously inferred loop invariants, size relations, and ranking functions for Java Card programs. However, they do not have a formal cost semantics and do not prove the bounds correct with respect to a cost model. Blazy et al. [11] have verified a loop bound analysis for CompCert's RTL intermediate language. However, this automatic bound analysis does not compute symbolic bounds.

11. Conclusion

We have developed a novel analysis framework for compositional and certified worst-case resource bound analysis for C programs. The framework combines ideas from existing abstract interpretation–

based techniques with the potential method of amortized analysis. It is implemented in the publicly available tool C^4B. To the best of our knowledge, C^4B is the first tool for C programs that automatically reduces the derivation of symbolic bounds to LP solving.

We have demonstrated that our approach improves the state-of-the-art in resource bound analysis for C programs in three ways. First, our technique is naturally compositional, tracks size changes of variables, and can abstractly specify the resource cost of functions (Section 3). Second, it is easily combinable with established qualitative verification to guide semi-automatic bound derivation (Section 6). Third, we have shown that the local inference rules of the derivation system automatically produce easily checkable certificates for the derived bounds (Section 7). Our system is the first amortized resource analysis for C programs. It addresses the long-standing open problem of extending automatic amortized resource analysis to compute bounds for programs that loop on signed integers and to deal with non-linear control flow.

This work is the starting point for several projects that we plan to investigate in the future, such as the extension to concurrency, better integration of low-level features like memory caches, and the extension of the automatic analysis to multivariate resource polynomials [25].

Acknowledgments

We thank members of the FLINT team at Yale and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by NSF grants 1319671 and 1065451, DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, and ONR Grant N00014-12-1-0478. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

[2] E. Albert, R. Bubel, S. Genaim, R. Hähnle, and G. Román-Díez. Verified Resource Guarantees for Heap Manipulating Programs. In *Fundamental Approaches to Software Engineering - 15th Int. Conf. (FASE'12)*, pages 130–145, 2012.

[3] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*, pages 117–133, 2010.

[4] D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In *19th Int. Static Analysis Symposium (SAS'12)*, pages 405–421, 2012.

[7] R. Atkey. Amortised Resource Analysis with Separation Logic. In *19th Euro. Symp. on Prog. (ESOP'10)*, pages 85–103, 2010.

[8] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal Certification of Code-Based Cryptographic Proofs. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 90–101, 2009.

[9] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie. System-Level Non-Interference for Constant-Time Cryptography. *IACR Cryptology ePrint Archive*, 2014:422, 2014.

[10] R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*, pages 103–118, 2010.

[11] S. Blazy, A. Maroneze, and D. Pichardie. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th Int. Conf. (VSTTE'13)*, 2013. To appear.

[12] V. A. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *7th Int. Symp. on Memory Management (ISMM'08)*, pages 141–150, 2008.

[13] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*, pages 140–155, 2014.

[14] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *28th Conf. on Object-Oriented Prog., Sys., Langs., and Appl., OOPSLA'13*, pages 33–52, 2013.

[15] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-End Verification of Stack-Space Bounds for C Programs. In *Conf. on Prog. Lang. Design and Impl. (PLDI'14)*, page 30, 2014.

[16] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional Certified Resource Bounds (Extended Version). Technical Report YALEU/DCS/TR-1505, Dept. of Computer Science, Yale University, New Haven, CT, April 2015.

[17] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX Annual Technical Conference (USENIX'10)*, 2010.

[18] M. Cohen, H. S. Zhu, E. E. Senem, and Y. D. Liu. Energy Types. In *27th Conf. on Object-Oriented Prog., Sys., Langs., and Appl., OOPSLA'12*, pages 831–850, 2012.

[19] COIN-OR Project. CLP (Coin-or Linear Programming). `https://projects.coin-or.org/Clp`, 2014. Accessed: 2014-11-12.

[20] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*, pages 292–304, 2010.

[21] S. Gulwani, S. Jain, and E. Koskinen. Control-Flow Refinement and Progress Invariants for Bound Analysis. In *Conf. on Prog. Lang. Design and Impl. (PLDI'09)*, pages 375–385, 2009.

[22] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*, pages 127–139, 2009.

[23] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *19th Euro. Symp. on Prog. (ESOP'10)*, 2010.

[24] J. Hoffmann and Z. Shao. Type-Based Amortized Resource Analysis with Integers and Arrays. In *12th International Symposium on Functional and Logic Programming (FLOPS'14)*, 2014.

[25] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th ACM Symp. on Principles of Prog. Langs. (POPL'11)*, 2011.

[26] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 2012.

[28] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on Principles of Prog. Langs. (POPL'03)*, pages 185–197, 2003.

[29] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th Euro. Symp. on Prog. (ESOP'06)*, pages 22–37, 2006.

[30] M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Joint 25th RTA and 12th TLCA*, 2014.

[32] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Emb. Sys., 11th Int. Workshop (CHES'09)*, pages 1–17, 2009.

[34] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[37] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4):751–778, 2005.

[38] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Approach for Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*, page 743–759, 2014.

[39] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[40] F. Zuleger, M. Sinn, S. Gulwani, and H. Veith. Bound Analysis of Imperative Programs with the Size-change Abstraction. In *18th Int. Static Analysis Symposium (SAS'11)*, 2011.

A. Complete Experimental Results for the Tool Comparison

Table 3. Comparison of the bounds generated by KoAT, Rank, LOOPUS, SPEED, and our tool C^4B on several challenging linear examples. Results for KoAT and SPEED were extracted from previous publications [20–22, 38] because KoAT cannot take C programs as input in its current version and SPEED is not available. Entries marked with ? indicate that we cannot test the respective example with the tool. Entries marked with — indicate that the tool failed to produce a result. We write $\mathrm{mx}(a, b)$ for the maximum of a and b. Functions with names of the form tXX are challenging tests that we designed during the development of C^4B. The source code for all functions is available in the extended version [16].

Function	KoAT		Rank		LOOPUS		SPEED	C^4B
gcd	?		$(((2{+}1)\dots$	$O(n)$	—		?	$\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
kmp	?		$(((2{+}(n{+}\dots$	$O(n^2)$	$\mathrm{mx}(n,0)\dots$	$O(n)$	?	$1{+}2\lvert[0,n]\rvert$
qsort	?		—		—		?	$1{+}2\lvert[0,\mathsf{len}]\rvert$
speed pldi09 fig4 2	—		$(((2{+}n)\dots$	$O(n)$	—		$\frac{n}{m}+n$	$1{+}2\lvert[0,n]\rvert$
speed pldi09 fig4 4	—		$(((2{+}(-1\dots$	$O(n)$	—		$\frac{n}{m}+m$	$\lvert[0,n]\rvert$
speed pldi09 fig4 5	$28d+7g+27$	$O(n)$	$(((2{+}(-1\dots$	$O(n)$	—		$\mathrm{mx}(n,n-m)$	—
speed pldi10 ex1	—		—		—		n	$\lvert[0,n]\rvert$
speed pldi10 ex3	—		$(((2{+}(-1\dots$	$O(n)$	$2{\cdot}\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
speed pldi10 ex4	$110a+33$	$O(n)$	—		—		$n+1$	$1{+}2\lvert[0,n]\rvert$
speed popl10 fig2 1	$9a+9b+\dots$	$O(n)$	$((2{+}((-y\dots$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$\lvert[x,n]\rvert{+}\lvert[y,m]\rvert$
speed popl10 fig2 2	$6a{+}9b{+}3c+5$	$O(n)$	$((2{-}x\dots$	$O(n)$	$\mathrm{mx}(0,(x+1{-}z)\dots$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,n{-}z)$	$\lvert[x,n]\rvert{+}\lvert[z,n]\rvert$
speed popl10 nested multiple	—		$((2{-}x{+}n\dots$	$O(n^2)$	$\mathrm{mx}(0,m{-}y)+\mathrm{mx}(0,n{-}x)$	$O(n)$	$\mathrm{mx}(0,n{-}x)+\mathrm{mx}(0,m{-}y)$	$\lvert[x,n]\rvert{+}\lvert[y,m]\rvert$
speed popl10 nested single	$48b+16$	$O(n)$	$(((1{-}x{+}n\dots$	$O(n)$	$\mathrm{mx}(0,n{-}1)\dots$	$O(n)$	n	$\lvert[0,n]\rvert$
speed popl10 sequential single	$21b+6$	$O(n)$	$((2-x{+}n\dots$	$O(n)$	$2{\cdot}\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
speed popl10 simple multiple	$9c+10d+7$	$O(n)$	$((2{-}y{+}m\dots$	$O(n)$	$\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$	$O(n)$	$n+m$	$\lvert[0,m]\rvert{+}\lvert[0,n]\rvert$
speed popl10 simple single2	$20d+12c+17$	$O(n)$	—		$\mathrm{mx}(n,0)+\mathrm{mx}(m,0)$	$O(n)$	$n+m$	$\lvert[0,n]\rvert{+}\lvert[0,m]\rvert$
speed popl10 simple single	$4b+6$	$O(n)$	$((2{-}x{+}n\dots$	$O(n)$	$\mathrm{mx}(n,0)$	$O(n)$	n	$\lvert[0,n]\rvert$
t07	?		$2+x$	$O(n)$	$\mathrm{mx}(x,0)\dots$	$O(n)$	?	$1{+}3\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t08	?		$((2{+}z{-}y\dots$	$O(n)$	$\mathrm{mx}(0,y{-}2)\dots$	$O(n)$	?	$1.33\lvert[y,z]\rvert{+}0.33\lvert[0,y]\rvert$
t10	?		$((2{-}y{+}x\dots$	$O(n)$	$\mathrm{mx}(0,x{-}y)$	$O(n)$	?	$\lvert[y,x]\rvert$
t11	?		$((2{-}y{+}m\dots$	$O(n)$	$\mathrm{mx}(0,n{-}x))+\mathrm{mx}(0,m{-}y)$	$O(n)$	?	$\lvert[x,n]\rvert{+}\lvert[y,m]\rvert$
t13	?		$(((1{+}y^2/2\dots$	$O(n^2)$	$2{\cdot}\mathrm{mx}(x,0)+\mathrm{mx}(y,0)$	$O(n)$	?	$2\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t15	?		$((1{+}x\dots$	$O(n)$	—		?	$\lvert[0,x]\rvert$
t16	?		$((-99{\cdot}y\dots$	$O(n)$	—		?	$101\lvert[0,x]\rvert$
t19	?		$((153{+}k\dots$	$O(n)$	$\mathrm{mx}(0,i{-}10^2){+}\mathrm{mx}(0,k{+}i{+}51)$	$O(n)$	?	$50{+}\lvert[-1,i]\rvert{+}\lvert[0,k]\rvert$
t20	?		$(2{-}y{+}x\dots$	$O(n)$	$2{\cdot}\mathrm{mx}(0,y{-}x){+}\mathrm{mx}(0,x{-}y)$	$O(n)$	?	$\lvert[x,y]\rvert{+}\lvert[y,x]\rvert$
t27	?		—		$10^3\mathrm{mx}(0,-n)\dots$	$O(n)$	?	$0.01\lvert[n,y]\rvert{+}11\lvert[n,0]\rvert$
t28	?		$((1{-}y{+}x\dots$	$O(n)$	$10^3\,\mathrm{mx}(0,x-y)\dots$	$O(n)$	?	$\lvert[x,0]\rvert{+}\lvert[0,y]\rvert{+}1002\lvert[y,x]\rvert$
t30	?		—		—		?	$\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t37	?		—		—		?	$3{+}2\lvert[0,x]\rvert{+}\lvert[0,y]\rvert$
t39	?		—		—		?	$1.33{+}0.67\lvert[z,y]\rvert$
t46	?		—		—		?	$\lvert[0,y]\rvert$
t47	?		$4+n$	$O(n)$	$1+\mathrm{mx}(n,0)$	$O(n)$	?	$1{+}\lvert[0,n]\rvert$